©Copyright 2019
John Toman

Learning to Adapt:
Analyses for Configurable Software

John Toman

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Daniel Grossman, Chair

Zachary Tatlock

Rastilav Bodik

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Learning to Adapt:
Analyses for Configurable Software

John Toman

Chair of the Supervisory Committee:
Professor Daniel Grossman
Computer Science & Engineering

Configurations are powerful tools for end-user customization of software. For example, non-expert software users may customize the behavior of programs via option menus, system administrators may tune server behavior by editing configuration files, and software developers may specialize generic frameworks to suit their purposes with software annotations and configuration files. Although configurations are extremely powerful tools that increase software flexibility, they increase implementation complexity causing subtle bugs and confounding analysis tools.

This dissertation considers the challenges posed by highly-configurable software and proposes that specialized program analyses can overcome these challenges. In particular, this dissertation explores two main topics. The first direction is creating program analyses to find defects in software that supports configuration changes at runtime. The second is the development of principled techniques for analyzing applications built on highly-configurable frameworks. These two research efforts are supported with formal proofs and empirical evaluations of proof-of-concept implementations.

TABLE OF CONTENTS

Page

Acknowledgments iv

Dedication vi

Chapter 1: Introduction and Thesis 1
- 1.1 Thesis Statement 3
- 1.2 Contributions 5
- 1.3 Structure of This Document and Relationship to Existing Work 7

Chapter 2: Dynamic Configuration Update Correctness and Dynamic Checking . 9
- 2.1 Introduction 9
- 2.2 Correctness Conditions for Dynamic Configuration Updates 11
- 2.3 Technique 15
- 2.4 Automated Bug Avoidance and Repair 27
- 2.5 Implementation 31
- 2.6 Evaluation 36
- 2.7 Conclusions 51

Chapter 3: Static Verification of External Resource Consistency 53
- 3.1 Introduction 53
- 3.2 At-Most-Once Problems 56
- 3.3 The Legato Analysis 58
- 3.4 Interprocedural Analysis 70
- 3.5 Implementation and Challenges 81
- 3.6 Evaluation 89
- 3.7 Conclusion 101

Chapter 4: Related Work for Dynamic External Resources 102

Chapter 5: Whole-Program Static Analysis of Modern Applications 107
5.1 Introduction 107
5.2 Static Analysis Challenges 108
5.3 Conclusion 118

Chapter 6: Hybrid Mostly-Concrete and Abstract Interpretation 120
6.1 Introduction 120
6.2 Overview 124
6.3 Preliminary Definitions 131
6.4 Combined Interpretation 138
6.5 Procedures 152
6.6 Iteration Strategy 156
6.7 Widening and Finitization 157
6.8 Extensions for a Realistic Prototype 159
6.9 Evaluation 167
6.10 Conclusion 171

Chapter 7: Supporting the Full JVM 172
7.1 Introduction 172
7.2 Java Path Finder 174
7.3 Current Progress 176
7.4 Scheduling Execution 179
7.5 Future Challenges and Potential Solutions 187
7.6 Conclusion 193

Chapter 8: Related Work in Whole-Program Static Analysis 194

Chapter 9: Conclusion and Future Work 202
9.1 Techniques 202
9.2 Future Directions 204

Bibliography 206

Appendix A: Proofs for Chapter 3 . . . 223
A.1 Preliminaries . . . 223
A.2 Concrete Instrumented Semantics . . . 224
A.3 Abstract Semantics . . . 225
A.4 Proof . . . 226

Appendix B: Proofs for Chapter 6 . . . 229
B.1 Proofs for Section 6.4 . . . 229
B.2 Proofs For Section 6.5 . . . 237
B.3 Formalisms and Proofs for Subfixpoint Iteration . . . 239
B.4 Formalisms, Soundness, and Termination of Widening Iteration . . . 257

Appendix C: Proofs for Chapter 7 . . . 267
C.1 Loop Consistency . . . 267
C.2 Proofs . . . 268

ACKNOWLEDGMENTS

This dissertation and the body of work it presents is the product of many different contributions from multiple people; I owe so much to so many people, and could fill pages and pages with acknowledgments and thanks.

My advisor Dan Grossman took a chance on me, and has shepherded me through grad school despite hard right turns into program analysis and abstract interpretation. He has been a steadfast source of paper feedback, reasoned advice, and his own peculiar brand of advisor encouragement (documented in many Facebook posts). I owe this PhD to him, and am eternally grateful for all he has done for me.

The PLSE lab at UW at different times provided valuable rubber ducking, coffee, at-large advising, paper editing, and proof checking. In particular, I'd like to thank James Bornholt for his advice, mentoring, and perspective that was always available when others were asleep. James Wilcox deserves acknowledgment for being my resident math consultant, who was willing to sit through long whiteboard proofs, and has patiently explained whether least upper bound operators are monotone.[1] Further thanks are due, in no particular order, to: Jared Roesch, Chandrakana Nandi, and Pavel Panchekha. I would also like to acknowledge those who took the time to make my papers better with thoughtful feedback; the work presented here is better thanks to them. In addition to all of the people thanked above, I would like to acknowledge: Ben Hardekopf, Bill Harris, Emina Torlak, Manuel Fahndrich, Francesco Logozzo, and Christian Kästner. I would also like to thank Jonathan Bell and Johannes Späth for their technical assistance during the Staccato and Legato projects. I would also like to thank my thesis committee for their selfless service. Reading this dissertation and sitting for exams was quite the time investment, and they all served with distinction.

I would like to particularly acknowledge my graduate student mentor, Doug Woos. Doug is not only a good friend (and colleague), but provided invaluable feedback on proofs, fellowship applications and paper drafts, and served as a critical lifeline during a difficult transition to graduate school life. He helped me manage expectations, navigate advisor relationships, and deal with the frustration of paper rejection. He is extremely generous with his time and advice; my success in graduate school is thanks to him.

Enough good things cannot be said about Professor Doctor Zachary Tatlock, PhD: his generosity with this time, energy, and enthusiasm is matched only by his heartfelt,

[1] They are not.

genuine desire to see students succeed, even when they aren't his own. Zach was my safety net and trusted confidant who always found time in his busy schedule for me. He is a model advisor, and I hope to be half as good as him someday.

Many thanks are due to my family for the various forms of support they gave during my time in graduate school, and for their tireless work to get me to where I am today. Special thanks are due to my late grandfather, Robert Toman: his frugality and planning gave me the opportunity to pursue higher education. I only wish he was here to see the results of his efforts.

I would like thank the friends that made my time in Seattle so rewarding: Ryan Maas, Camille Cobb, Jeff Snyder, Daryl Zuniga, Joe Redmon, Aleks Holynski, and Harley Montgomery. Some of my best memories are going places and losing board games to these people.

Although it will be many years until he can read this, I want to thank my son Corvin Tobias Beswick Toman for all the joy he brings to my life. But most of all, I want to thank my wife, Laura Toman. Laura's endless and tireless support made all of this possible; I owe her everything. She was my partner, confidant, support, best friend, and board game opponent. She kept our family and lives on track while I finished this dissertation. This dissertation is 274 pages long; I could easily fill twice that many with her praises. Thank you Laura, you made this all worth it.

DEDICATION

Dedicated to my wife, Laura Toman, without whom none of this would be possible. This dissertation and the work contained within is as much a product of her hard work as it is mine.

ACCESSING THIS DISSERTATION

This dissertation was produced under the spirit of open scholarship. Please do not pay for access to this document: if you cannot find a freely available PDF of this dissertation, contact me at jtoman@cs.washington.edu. I will happily provide a copy of this work free of charge.

Chapter 1

INTRODUCTION AND THESIS

Today's software is highly configurable [207, 208, 97]. Configurations are found across the software stack, from kernel compile-time options [143], command-line options [176], configuration menus in desktop software [96] and mobile applications [120], to server and cluster software configuration files [202]. These configuration options[1] increase the portability and reusability of software. By tweaking the right configuration options, an end-user can modify a program's behavior in a controlled manner to better suit his or her needs. For example, the number of threads used by a web server can be increased in response to spikes in web traffic, or unneeded kernel features can be disabled for performance.

In addition to user-facing configuration options, configurations are ubiquitous in the realm of software engineering. Modern applications are rarely written from the ground up; developers use existing, proven libraries and frameworks to handle basic, common functionality. Like user-facing applications, developers can configure libraries and frameworks to better suit their requirements. For example, almost every major component of Spring (a Java application framework) can be tweaked or customized using XML files and/or program annotations.

Without configuration options, modifying program behavior would be solely the domain of software engineering experts. For instance, changing the number of threads used by a webserver would require:

1. Finding the source code of the webserver,

[1] A note on vocabulary: I am unaware of any standard, rigorous definition that clearly differentiates between a configuration option and program input. For this document, I will leave the meaning of "configuration option" ambiguous, relying on the reader's intuitive understanding of the term.

2. Identifying the source code file/module responsible for thread management

3. Changing the number of threads, including rigorous testing to ensure any changes do not introduce new errors,

4. Recompilation of the modified program, and finally

5. Redeployment, which may include repackaging, archive signing, distribution, etc.

Several things must go right for the above process to succeed. At the very least, the source code of the webserver must be available (an impossibility if the server is proprietary) and the user must be both an expert in the implementation language *and* the program itself. In contrast, armed with a well-documented and easily accessible configuration interface, the end-user may effect the desired change by modifying a single number. Further, most industrial-strength software supports configuration changes at runtime, obviating redeployment and lost uptime. Similarly, if frameworks did not allow extensive customization via configuration files and annotations they would be far less convenient for software developers

Software configurations effectively allow modification of a program's behavior without modifying its source code *after* it has been delivered to the end user.[2] As such, they are extremely powerful tools, enabling a (limited) form of program customization with end-user programming.

Unfortunately, the convenience of configurable software does not come without cost. Continuing with the programming metaphor, configuration options, like any programming language, can be used incorrectly. End-users, faced with a dizzying array of configuration options, can misconfigure programs leading to crashes or performance issues.

---

[2]This characterization does not hold universally, as kernel configuration options *do* require changing source code and recompilation. This edge case illustrates the difficulty of precisely defining a configuration option.

Similarly, developers must also reason about non-obvious interactions between configuration options, and must maintain several different variants of the same programs. Even implementing configuration updates is challenging, as a recent bug[3] in Solr[4] demonstrates; this bug prevented Solr from correctly applying a configuration change requested by the user. Finally, configurable frameworks exhibit an enormous number of possible behaviors which depend on application-specific configurations. This flexibility is often implemented using difficult-to-analyze features such as reflection, metaprogramming, and multiple layers of abstraction. The framework effectively functions as an interpreter for some configuration language. Static analysis developers must contend with this extreme variability, pervasive use of difficult-to-analyze idioms, and lack of information when analyzing framework-based applications. Analyzing a framework implementation without considering the configuration information is roughly equivalent to guessing the behavior of an interpreter's execution *without* access to the source program that produced that execution.

The need for configuration-aware analyses is clear given this modern software landscape and the challenges it poses. Some steps have already been taken in this direction. For example, many researchers have investigated helping users diagnose crashes or performance problems due to misconfiguration [207, 197, 201, 9, 10, 96, 208, 67, 158]. Other researchers have helped developers understand the implications of configuration options [120, 161], and developed techniques to exploit static framework configurations to improve static analysis precision and soundness [8, 189, 174]. Despite this progress, significant challenges remain in the field of configurable software.

1.1 Thesis Statement

I propose that **the quality of modern, highly-configurable software can be improved with program analyses and techniques that reason directly about software configura-**

---

[3] https://issues.apache.org/jira/browse/SOLR-3587

[4] http://lucene.apache.org/solr/

tions. Under this broad thesis I will explore and substantiate two more specific theses. In this section I state these theses and their meaning; the contributions which substantiate these claims are described in the next section.

The first thesis posits that **defects in programs' support for configuration changes at runtime can be detected with static and dynamic techniques.** In particular, this thesis concerns programs that support configuration changes which take effect *without* program restart or service interruption. These update implementations must contend with concurrency, aliasing, and caching; failing to account for any one of these may cause program errors. This thesis claims that specialized analyses can effectively detect these errors. Within this thesis is the implicit claim that we can characterize the correctness of these implementations and configuration changes in a formal, uniform way.

The second thesis states that **the performance and precision of static analyses over framework-based applications can be improved with the principled combination of abstract and concrete interpretation that also exploits static configuration information.** Recall that frameworks are both highly configurable and use difficult-to-analyze program features that depend on application-specific configurations. Existing static program analysis techniques generally must sacrifice some combination of performance, precision, or soundness. In other words analyses may either miss errors (soundness), give too many incorrect answers (precision), or take unreasonably long to execute (performance). This thesis claims that concretely executing framework code while using abstract interpretation on the application code is a precise and performant analysis strategy. In particular, it posits that the concrete interpreter may exploit a framework's configuration information to precisely and exactly resolve difficult-to-analyze program features. Further, this thesis claims that this combination can be proved sound, i.e., it is a principled approach to combining interpretation strategies.

1.2 Contributions

To substantiate the above theses, this dissertation presents three major contributions. I now expand on these contributions and provide the high-level results and takeaways.

Staccato STACCATO is a dynamic analysis which finds defects in programs' support for configuration changes that take effect at runtime without service interruption. I hypothesized that these updates (which we call "dynamic configuration updates" or DCU) are difficult to implement correctly, and failing to do so may cause subtle and difficult to diagnose program errors. STACCATO is, to the best of my knowledge, the first work to address this problem. I first developed two novel correctness conditions for DCU implementations and hypothesized that violations of these correctness conditions would cause undesirable program behavior. I then implemented the STACCATO analysis, which detects these violations by monitoring executions of instrumented Java programs. I confirmed STACCATO's effectiveness with an empirical evaluation of our correctness conditions and analysis on 3 real-world Java programs. STACCATO detected violations of the correctness conditions in all of the Java programs, and confirmed with manual inspection that these violations corresponded to (sometimes serious) program errors. These experimental results indicate that some programmers do struggle to implement DCU correctly, and STACCATO is an effective tool for finding bugs in program's dynamic configuration update functionality. In summary, the contribution of STACCATO is **an empirically effective dynamic analysis which detects violations of novel correctness conditions for online configuration update mechanisms**.

Legato LEGATO is a static approach for finding defects in dynamic configuration update implementations. To make detecting these defects tractable in a static setting, I developed a third, novel correctness condition called the *at-most-once* condition. This condition over-approximates one of the correctness conditions checked by STACCATO. It

is amenable to static checking, unlike the two correctness conditions used by STACCATO. I developed a sound, heap-, context-, flow-, and field-sensitive static analysis which verifies that a program adheres to the at-most-once condition, and by extension, does not contain certain classes of DCU errors. In fact, as argued in our work, the at-most-once condition is appropriate for verifying consistent interaction with parts of a program's dynamically changing execution environment beyond just configuration options. We evaluated LEGATO on 10 real-world Java applications and found 65 violations of the at-most-once condition which correspond to real program defects in dynamic configuration update implementations. This result further establishes that programmers struggle to correctly implement DCU mechanisms and that LEGATO is an effective static analysis for detecting these errors. In summary, LEGATO is **an empirically validated static analysis which checks the novel *at-most-once* condition which can find errors in dynamic configuration update implementations**. Further, the work on LEGATO and STACCATO are the core contributions which substantiate my first thesis.

Concerto Finally, this dissertation presents my work on CONCERTO, an analysis framework designed to increase analysis performance and precision on framework-based applications. CONCERTO combines concrete and abstract interpretation by concretely executing framework code and running an abstract interpreter on application code. By combining analysis strategies, CONCERTO benefits from the strengths of both techniques: the scalability and computability of abstract interpretation plus the precision of concrete interpretation. During concrete execution, CONCERTO reads a framework's static configuration, which often ensures the framework takes a single, deterministic path of execution. My work formalized this hybrid model within the theory of abstract interpretation and proved that the combination of abstract and concrete interpretation soundly summarizes program behavior. I also showed that CONCERTO delivers provably improved precision for certain classes of abstract interpretation. I developed a prototype implementation of CONCERTO which analyzes a Java-like, object-oriented language. To confirm the benefits

of combined interpretation, I applied this prototype to a difficult-to-analyze, highly-configurable framework. The prototype of Concerto was significantly more precise and performant compared to standard abstract interpreters. In summary, Concerto is a **principled framework for precisely analyzing framework based applications with the combination of concrete and abstract interpretation.** This work substantiates the second thesis above.

1.3 Structure of This Document and Relationship to Existing Work

I now outline the remainder of this dissertation. Most of the work presented in this dissertation has been published in conference proceedings. Where appropriate, I will note the differences between the material presented here and past papers.

The first half of this dissertation (Chapters 2–4) focuses on the first thesis. Chapter 2 describes the Staccato dynamic analysis sketched above. This chapter is based upon work published in ECOOP 16 [182] and contains a discussion of consistency groups that were omitted from the original paper. Chapter 3 presents the Legato analysis and is an extended version of the paper published in ECOOP 18 [184]. I have expanded the termination argument, described the representation of our edge functions, and provided a closer comparison with Staccato's results and techniques. Finally, Chapter 4 reviews the related work of Legato and Staccato.

The second half of this dissertation (Chapters 5–8) focuses on the second thesis. Chapter 5 describes the challenges found in analyzing modern applications and motivates the following two chapters. This chapter is an extended version of the first half of a position paper published in SNAPL 17 [183]. I have expanded this chapter to include concrete examples of difficulties I faced during the development of Legato. Chapter 6 describes Concerto and presents the results of our experiments. This chapter is based upon work published in POPL 19 [185], and has been extended with additional notation and examples that were omitted from the original version for space reasons. This chapter also corrects a notation error that appeared in our conference version. Chapter 7 describes

the in-progress work on extending Concerto to handle the full JVM and Java language. This project, named Symphony, is not yet complete and this work is unpublished. The chapter describes the current progress on Symphony, a novel analysis scheduling algorithm, the remaining technical and research challenges, and potential solutions to these challenges. Chapter 8 closes the second half with a discussion of related work in Concerto and Symphony's areas.

Finally, the dissertation concludes with Chapter 9, which summarizes the dissertation and describes some future work. Some of the directions outlined are drawn from my previously cited position paper [183]. Appendices A and B contain the proofs for Legato and Concerto respectively. Appendix C contains some preliminary correctness proofs of the scheduling algorithm described in Chapter 7.

Chapter 2

DYNAMIC CONFIGURATION UPDATE CORRECTNESS AND DYNAMIC CHECKING

2.1 *Introduction*

The configurability of modern software described in Chapter 1 increases the reusability and portability of software. Along with high levels of configurability, software is increasingly subject to stringent uptime requirements; restarting an application to effect a configuration change is not always feasible. To accommodate configurability and robustness, many applications support configuration options that can be changed at runtime. We call a configuration change at runtime a *dynamic configuration update* (DCU).

Unfortunately, it is difficult to implement dynamic configuration updates correctly. The Solr bug[1] mentioned in Chapter 1 demonstrates this challenge. The Solr user may specify *analyzers* in a configuration file that is read by Solr at startup. These analyzers process text before it analyzed, applying case normalization, removing stop words, and so on. Solr also provides a reload command which re-reads the configuration file and re-initializes the system. However, the analyzers were not updated to reflect the new configuration, even though Solr reported the reload complete. Solr would then silently misprocess data and incorrectly answer user queries. The only indication that something was wrong was Solr's output, which required careful inspection on the part of the user. We found similar defects in multiple applications.

This chapter presents, to the best of our knowledge, the first study of defects in dynamic configuration update implementations. We have developed a dynamic analysis that can assist developers in finding and diagnosing defects in their DCU implementa-

[1] https://issues.apache.org/jira/browse/SOLR-3587

tions. We implemented our technique in STACCATO (STAle Configuration and Consistency Analysis Tool), a prototype DCU error detection tool for Java programs.

Our approach checks one of two alternative correctness conditions for DCU systems, as chosen by the programmer. The first condition states that old versions of configuration options may not be used after a configuration update. The second states that only one version of a configuration option may be used during a single method execution. Both conditions provide a possible specification of program behavior in the presence of dynamic configuration updates. Choosing the appropriate correctness condition for a program unit requires domain knowledge. We have the user of STACCATO select the appropriate condition with a few lines of high-level annotations (fewer than 0.7 per 1,000 SLOC in our evaluation).

Underlying both conditions is the notion of *versioning* the software configuration. For every program value, STACCATO tracks which configuration options, and which versions of those options, were used to construct that value. This information moves with a value as it transformed and combined with other values. Whenever a value is read by the program, STACCATO checks that the value does not violate the correctness condition selected by the programmer. Our analysis supports concurrency and requires no changes to the JVM.

In addition to bug finding, STACCATO provides support for program repair and bug avoidance. STACCATO can transparently repair program schedules to hide insufficient synchronization around configuration accesses. In addition, when STACCATO detects a value built from out-of-date configuration options, it calls a programmer-provided callback to update the stale value. We used this functionality to repair DCU errors that we discovered in our evaluation. We also found this technique to be effective enough that for some projects, we were able to add DCU support for options that previously required a restart.

We applied STACCATO to three large open-source projects with extensive support for dynamic configurability. These were code-bases with which we were not previously

familiar, but we were still able to effectively use Staccato to find DCU defects in all of them.

In summary, this chapter makes the following contributions:

- We provide two correctness conditions for software that supports dynamic configuration updates (DCU).
- We define the problem of detecting and diagnosing errors in DCU code.
- We describe how errors in DCU implementations can be discovered with a novel dynamic information-flow tracking approach.
- We describe Staccato, a tool implementing this approach for Java programs.[2]
- Using Staccato, we show that bugs in DCU implementations affect multiple open-source projects, and that our technique is effective for finding these errors.

The rest of this chapter is organized as follows. Section 2.2 introduces and justifies the correctness conditions checked by our approach. Section 2.3 introduces an information-flow tracking technique for detecting correctness violations in programs that support DCU. Section 2.4 details our approach for bug avoidance and program repair. Section 2.5 describes our implementation. Section 2.6 presents the results from applying Staccato to three open-source software projects. Section 2.7 concludes.

2.2 *Correctness Conditions for Dynamic Configuration Updates*

Staccato identifies errors caused by defects in dynamic configuration update mechanisms. We are unaware of any agreed upon correctness definition for DCU implementations either in industry or the research literature (see Chapter 4). Accordingly, we

[2] Staccato is open-source. Our implementation and the tests we ran for this research can be found at https://github.com/uwplse/staccato

```java
class RequestManager implements Reloadable {
  String targetIp, apiKey;
  RequestManager() {
     targetIp = Config.get("request.target-ip");
     apiKey = Config.get("request.api-key");
  }
  String doRequest() {
     Request req = new ApiRequest(targetIp);
     req.send(apiKey);
     return req.response();
  }
  void reloadConfig() {
     targetIp = Config.get("request.target-ip");
  }
}
```

Figure 2.1: A bug caused by an incomplete configuration update: after an update `reloadConfig()` does not read the updated `"request.api-key"` option.

```java
class SpotService implements WebService {
  String handleSpotRequest() {
    return Config.get("verb") + " Spot, " + Config.get("verb") + "!";
  }

  void handleUpdateRequest(String newVerb) {
    Config.set("verb", newVerb);
  }
}
```

Figure 2.2: A bug caused by an inconsistent view of the configuration: the `handleSpotRequest` method can observe two versions of the `"verb"` option in one execution.

first give two correctness definitions for DCU schemes that we developed after surveying configurable software systems. Both conditions provide a specification for program behavior in the presence of DCU. The programmer selects which condition to use for an application, with the option of switching conditions (or opting out altogether) on a per-method or per-class basis with annotations.

We begin with two motivating examples found in Figures 2.1 and 2.2. Both examples are invented code fragments that are representative of two common errors we found during our survey of configurable software. The `Config` class provides a mapping of configuration options to string values. In Figure 2.1, the user may update either the target IP or API key configuration options, after which the `reloadConfig()` method is called. The implementation has an error in its handling of the update: the `reloadConfig()` method fails to read the new value of the `"request.api-key"` configuration option. As a result, the configuration update requested by the user is not completely applied. The Solr bug described in the introduction is similar to this example.

Figure 2.2 shows a consistency issue that we frequently found in configurable software. In this example, `SpotService` implements a web service that can be configured with an administrative command. The primary functionality, expressed in the `handleSpotRequest` method, is to simply echo back sentences of the form, `"Run Spot, Run!"`. The verb in the response is controlled with the `"verb"` configuration option. An administrator can update the `"verb"` option by sending an update request that is handled by the `handleUpdateRequest` method. If an update request is received concurrently with a regular client request, a user can receive unexpected responses such as `"Run Spot, Bark!"` This error occurs because the use of `"verb"` in `handleSpotRequest` is not *atomic*: a single invocation of `handleSpotRequest` can observe two inconsistent versions of the `"verb"` configuration option.

Guided by these two examples and many others like them found in real programs, we have developed two conditions for software that supports DCU. The first condition is as follows:

Correctness Condition C1 (*Staleness*): An execution must observe only values derived from the most *up-to-date* version of the program configuration.

We call correctness condition C1 the *staleness* condition as it states that a program may not use values built from stale configurations. This correctness condition is potentially violated in both examples. In the first example, after a configuration update the program will read a stale version of the `"request.api-key"` option on line 9 in Figure 2.1. In the second example, the string returned by `handleSpotRequest` may be derived (partially or completely) from an old version of the `"verb"` option.

While condition C1 is sufficient to identify the bug in Figure 2.2, it is overly strict. For example, consider again the scenario where the service in Figure 2.2 receives two concurrent requests handled by `handleSpotRequest` and `handleUpdateRequest` respectively. Due to non-determinism in thread scheduling, `handleSpotRequest` may return the response `"Run Spot, Run!"` after `handleUpdateRequest` has updated `"verb"` to another value, such as `"Bark"`. This outcome violates condition C1, but some systems allow this behavior for high performance. A more desirable correctness condition would rule out only *inconsistent* outcomes, such as `"Run Spot, Bark!"`, which violate the programmer's expectations about configuration behavior. This observation motivates a second correctness condition:

Correctness Condition C2 (*Consistency*): A method execution may observe only a *single* version of each configuration option.

In other words, a program may use an old version of a configuration option provided that version is not mixed with any other versions of the same option.

Both correctness conditions are useful in different situations. Which condition is appropriate for a given program is application dependent. In practice, we found that condition C2 (consistency) was a reasonable default for the programs in our evaluation set. Nevertheless, the staleness condition is a good fit for global, configurable objects, such as caches, database connections, etc. This led to a useful rule of thumb for using our technique: use the staleness condition for methods that manipulate configurable objects

stored in static class members and use the consistency condition everywhere else. This was sufficient to find several bugs, and required fewer than 1 annotation per 1,000 SLOC across our evaluation set.

2.3 Technique

Staccato detects errors in DCU schemes with a dynamic information-flow analysis in the style of taint analysis. There are three pieces of Staccato's analysis, described in the next three subsections. First, Staccato versions the software configuration and associates each program value with a *configuration history* (Section 2.3.1). A value's history records which options, and what *versions* of those option, were used to build that value. Second, when values are combined, their configuration histories are merged to produce a history for the output value (Section 2.3.2). This *propagation* models the flow of configuration information through a program. Finally, whenever a value is read by the program, Staccato checks for violations of the DCU correctness conditions (Section 2.3.3). To ease explication, we will first describe our technique in terms of only detecting violations of the staleness condition (C1). Our approach for the consistency condition (C2) is presented in Section 2.3.4 as an extension to this technique.

2.3.1 Configurations Values

Staccato models the configuration of a program as a global map from strings to strings. Following standard terminology, we call each key an *option*. We call each value in the map a configuration *value*. Staccato also internally associates each configuration option with an *epoch* counter. These option epochs *version* the configuration values of a program. The epoch counters are stored in a map $\mathcal{V}$ from options to epochs: the current epoch of an option is denoted $\mathcal{V}[o]$. When an option o is updated, the value of $\mathcal{V}[o]$ is incremented. We will use e to denote arbitrary epoch values.

Staccato also tags each program value with a *configuration history*. The configuration

```
1 String foo(String a, String b, boolean t) {
2   if(t) { return a; } else { return b; }
3 }
4 // 𝒱["foo"] = 2
5 String s = Config.get("foo"); // sᴴ = {"foo" → 2}
6 Config.set("foo", "..."); // 𝒱["foo"] = 3
7 String r = Config.get("foo"); // rᴴ = {"foo" → 3}
8 String q = foo(r, s, false); // qᴴ = {"foo" → 2}
9 q = foo(r, s, true); // qᴴ = {"foo" → 3}
```

Figure 2.3: Example configuration histories in a program. The history of the configuration value returned by `Config.get` on line 5 maps `"foo"` to the current version of `"foo"`, $\mathcal{V}[\texttt{"foo"}] = 2$. After the update on line 6, the epoch of `"foo"` is incremented and the value returned on line 7 is tagged with the new version. Notice that on lines 8 and 9 the configuration histories of `r` and `p` have moved through method parameters and return statements automatically.

history for a value v records the configuration options used to construct that value, along with the epochs (i.e., versions) of those options. The notation $v^{\mathcal{H}} = \{o_1 \to e_1, o_2 \to e_2, \ldots\}$ denotes that v was constructed using the value of configuration options o_1, o_2, ... at versions e_1, e_2, ... We will use $v^{\mathcal{H}}$ as a function mapping strings (option names) to epochs. $\mathrm{dom}(v^{\mathcal{H}})$ denotes the set of options that appear in the configuration set $v^{\mathcal{H}}$. The notation $x^{\mathcal{H}}$ may be used when x is a variable to denote the history of the value referenced by x.

Configuration histories are associated with program values (not textual program variables). Thus, as a value moves through field array, variable assignments, or method boundaries, its configuration history automatically moves with it. This is important as DCU bugs involve complex information-flow: the Solr bug from the introduction involved dataflow through 7 classes.

Configuration histories are empty for most values: only values that depend on the program configuration may have non-empty histories. Initially, the only values associated with non-empty configuration histories are configuration values themselves. When an option o is retrieved from the program's configuration, the returned configuration value is tagged with the history $\{o \to \mathcal{V}[o]\}$ (recall that $\mathcal{V}[o]$ is the current epoch of o). An example of this tagging and epoch counter updating can be found in Figure 2.3.

2.3.2 Propagation

Staccato tracks configuration values as they are combined with other values or transformed by the program. Thus, when multiple values are combined to create a new value, Staccato ensures that the new value's configuration history soundly captures all configuration information associated with the source values. The process of transferring configuration histories from operands to outputs is called *propagation*.

During propagation, configuration histories are merged together pairwise to yield the final configuration history of the output value. The merge operation for configuration

```
1 // V["foo"] = 2 and V["bar"] = 4
2 String f = Config.get("foo"); // f^H = {"foo" → 2}
3 String g = Config.get("bar"); // g^H = {"bar" → 4}
4 String h = f + g; // h^H = {"foo" → 2, "bar" → 4}
5 Config.set("foo", "..."); // V["foo"] = 3
6 String j = h + Config.get("foo"); // j^H = {"foo" → 2, "bar" → 4}
```

Figure 2.4: An example of history propagation and the history merge operation. On line 4, the input configuration histories have disjoint domains and therefore the merge operation is trivial. On line 6 the input configuration histories both have `"foo"` in their domains. According to the merge rule, the smaller of the two epochs for `"foo"` is chosen, which in this case is 2.

histories is denoted $v^{\mathcal{H}} \cup u^{\mathcal{H}}$. If $\mathrm{dom}(v^{\mathcal{H}}) \cap \mathrm{dom}(u^{\mathcal{H}}) = \emptyset$, then the merge operation is a simple union. However, it is possible that some option o appears in both $v^{\mathcal{H}}$ and $u^{\mathcal{H}}$. Staccato conservatively handles this situation. Suppose that $v^{\mathcal{H}}[o] = e$ and $u^{\mathcal{H}}[o] = e'$. The output configuration history maps o to the epoch $\mathrm{min}(e, e')$. This definition ensures that the epoch recorded for a configuration option o in a configuration history $x^{\mathcal{H}}$ is the oldest version of o used (either transitively or directly) to construct x. Figure 2.4 demonstrates history propagation and an example of the merge operation.

Staccato does not check for violations of the correctness condition during propagation. Recall that Staccato allows the programmer to select one of two possible correctness conditions or to opt out of checking altogether. If we integrated checking into propagation, we would have to implement three different propagation operations: one for each correctness condition and one for propagation without any checking. By separating the check and propagation operations, we can apply a uniform propagation operation across the entire program and vary the check operation depending on the correctness condition selected by the programmer.

```java
class DBConnection {
  @StaccatoPropagate(RETURN)
  static DBConnection connect(String host, String user) { ... }
  String fetchRow(String query) { ... }
}
User getUser(...) {
  String host = Config.get("db-host"), user = Config.get("db-user");
  DBConnection conn = DBConnection.connect(host, user);
  String userInfo = conn.fetchRow(...);
  return new User(userInfo);
}
```

Figure 2.5: Propagation involving objects dependencies. The `@StaccatoPropagate` annotation the `connect` method propagates the configuration histories of the `host` and `user` parameters to the `DBConnection` object returned from the method. As a result, the configuration history of the `conn` in the `getUser` method has the `"db-host"` and `"db-user"` options in its configuration history.

Propagation for Object Types A key design decision we faced is where propagation should occur. Operations such as integer arithmetic, boolean operations, and string concatenation all naturally propagate configuration information from source operands to output values automatically. Although sufficient for tracking simple dependencies, propagation involving only primitive values will miss dependencies at the object level. Consider the example in Figure 2.5. In order to verify that the usage of the connection object created on line 8 is correct, `conn` must inherit the configuration information of the `host` and `user` configuration values. Therefore, Staccato also supports propagation from method arguments to outputs. This method-level propagation expresses configuration dependencies at a higher level of abstraction than what is possible with just primitive operations.

The user opts-in to method-level propagation on a per-method basis with annotations like the one seen on line 2 in Figure 2.5. The annotation argument indicates which output of the method is the target of propagation. There are two possible outputs: the return value (specified with the `RETURN` argument seen in the example) or the method receiver. The receiver output is used primarily for modeling side-effecting methods such as setters or constructors. After an annotated method is executed, Staccato combines the configuration histories of the method's arguments with the existing history of the receiver or return value as appropriate. For the purpose of propagation to the return value, the method receiver (if applicable) is treated as an argument. In the database example, the propagation annotation ensures that $\texttt{conn}^{\mathcal{H}}$ contains the `"db-user"` and `"db-host"` options.

We experimented with an eager approach that propagated configuration histories unconditionally from method arguments to method receiver/return values. This seemed reasonable, as function outputs generally depend on input values. This strategy *would* properly capture the relationship between the `"db-host"` and `"db-user"` options and the `conn` object without user annotation. In practice, this approach suffers from a significant loss of precision. In Figure 2.5, automatic propagation also "taints" the value read

from the database on line 9 with configuration information. By extension, the `User` object created on line 10 would also contain the database configuration options in its configuration history. However, using a `User` object built with data read from an old connection is arguably *not* an error. This scenario is the expected behavior envisioned when a configurable database connection was implemented. In general, automatic propagation tags program data with unintuitive configuration histories, leading to false positives.

In our experience, configuration propagation is fairly rare; compared to the number of method definitions in our evaluation code-bases, the number of propagation annotations required was quite low, requiring fewer than 30 annotations across the entire evaluation set. We also found that propagation annotations need only be applied at natural API method boundaries, requiring no reasoning about whole program flow or API usage sites.

2.3.3 *Checking*

The final component of Staccato checks that the program does not observe values constructed from an out-of-date version of the configuration. Our approach actually checks an equivalent condition: that the value read by the program reflects only the most recent version of the configuration. A value v is up-to-date with respect to the current version of the configuration (as recorded in $\mathcal{V}$) iff: $\forall o \in \mathtt{dom}(v^{\mathcal{H}}), v^{\mathcal{H}}[o] = \mathcal{V}[o]$. In other words, only the most recent configuration values may have been used (transitively or directly) to construct v. This check is performed on value reads. For the purposes of our analysis a value is read when it is: 1) passed as an argument to a method, 2) read from a field, 3) returned from a method, 4) read from a local variable, or 5) read from an array. If Staccato detects that the condition has been violated, we have identified a DCU defect and alert the user.

2.3.4 *Extensions for Checking Consistency*

We now discuss how to extend our approach to check for violations of the consistency condition (C2). Recall that the second correctness condition states that a method execution may not observe inconsistent versions of the program configuration. Checking this condition requires extending the representation of configuration histories and the merge operator.

In addition to option epochs, STACCATO also tracks an extra bit for each option o in a configuration history. This bit is called the consistency flag. We denote this epoch-bit pair with $\langle e, f \rangle$, where e is an epoch and f is the new consistency flag. The consistency flag becomes set when STACCATO detects that different versions of the same option have been used to construct a value. Any values derived (in terms of the propagation described in Section 2.3.2) from an inconsistent value are themselves marked as inconsistent. The consistency flag is initially unset ($f = 0$) for configuration values returned from the program's configuration: by definition, a configuration value always represents a single version of a configuration option.

The definition of the consistency flag yields the following extension to the merge operator. Given two configuration histories to be merged, $u^{\mathcal{H}}$ and $v^{\mathcal{H}}$, such that $u^{\mathcal{H}}[o] = \langle e, f \rangle$ and $v^{\mathcal{H}}[o] = \langle e', f' \rangle$, the consistency flag in the merged configuration history is set iff: $e \neq e' \lor f = 1 \lor f' = 1$. The epoch for o is computed using the $\mathtt{min}$ function as previously described.

Given these extensions to configuration histories and the merge operation, checking the consistency condition is straightforward. The consistency condition implies that at most one version of each option may be used to build a value. The consistency flag precisely tracks this requirement. Thus, checking a value v against the consistency condition entails checking that all consistency flags in $v^{\mathcal{H}}$ are unset. That is, a value v reflects a consistent view of the configuration iff: $\forall o \in \mathtt{dom}(v^{\mathcal{H}}).v^{\mathcal{H}}[o] = \langle e, f \rangle \to f = 0$. This condition is checked at the same program points as the staleness condition.

```
1 int output = getValue();
2 while(...) {
3   if(Config.get("op") == "+") {
4     output += 2
5   } else {
6     output *= 2;
7   }
8 }
```

Figure 2.6: A simplified example of DCU error involving control-flow. If the "op" option is changed during the execution of the loop, the `output` variable will be processed inconsistently.

Control-Flow We have so far discussed the consistency condition in terms of *values* produced and read by a program. This misses inconsistent behavior introduced by control-flow. A simplified example, based on a real bug found in our evaluation set, is shown in Figure 2.6. In this example, if the "op" property is updated before the loop terminates, the `output` variable can reflect two different versions of the configuration. Lillack et al. [120] have noted that configuration values are often used in branching decisions so detecting errors involving control-flow is especially important.

To find errors like the one in Figure 2.6, we verify that the entire execution of a method observes a consistent view of the configuration. To check this property, upon entrance to a method we allocate a special sentinel object Σ. This object has a configuration history, $\Sigma^{\mathcal{H}}$, which is empty at method entry. Unlike regular configuration histories, which record which configuration options and versions were used to construct a *single* value, $\Sigma^{\mathcal{H}}$ records the options and versions used by the *entire* method. Whenever a value v is read (as defined in Section 2.3.3), $\Sigma^{\mathcal{H}}$ is updated such that $\Sigma^{\mathcal{H}'} = \Sigma^{\mathcal{H}} \cup v^{\mathcal{H}}$. $\Sigma^{\mathcal{H}'}$ is immediately checked for consistency using the rule described above. At method exit, Σ

and its configuration history are discarded; each invocation begins with a fresh history. Method histories are not inherited by callers or callees.

To see how this approach catches errors involving control-flow, consider again the example in Figure 2.6. Assume at the entrance to the loop $\Sigma^{\mathcal{H}}$ is initially empty. On the first iteration of the loop, the `Config.get("op")` call on line 3 returns a configuration value with the configuration history $\{$`"op"` $\rightarrow \langle 1, 0 \rangle\}$. This is merged with Σ's empty configuration history, giving $\Sigma^{\mathcal{H}} = \{$`"op"` $\rightarrow \langle 1, 0 \rangle\}$. At the end of the first iteration, `"op"` is updated. On the second iteration, the value returned by the `get()` call on line 3 is a newer version of the `"op"` option, which is tagged with the history $\{$`"op"` $\rightarrow \langle 2, 0 \rangle\}$. This history is merged with $\Sigma^{\mathcal{H}}$. The result of this merge is $\Sigma^{\mathcal{H}} = \{$`"op"` $\rightarrow \langle 1, 1 \rangle\}$. Notice that the consistency flag for `"op"` is now set: this signals that the execution has now observed two conflicting versions of the `"op"` option. Staccato immediately reports this as an error.

The assumption that a method execution is the single unit for consistency is a heuristic, which may admit false positives and false negatives. A more precise approach would require precise interprocedural tracking of control-flow dependencies. We experimented with a version of Staccato that integrated an existing off-the-shelf implementation of precise control-flow tracking found in recent versions of Phosphor [17]. Although this approach was promising on small test examples, it suffered scalability issues when applied to the programs in our evaluation set. For example, after applying Staccato with control-flow tracking to a web application, the program failed to return a response to the client after 5 minutes. This overhead is due to the large amount of state maintenance required for precisely tracking control-flow dependencies. In contrast, our approach is relatively cheap to compute and can find inconsistencies introduced through control-flow. In our experiments, this approach was not the source of any false positives.

Consistency Groups The approach described so far has assumed every configuration option is independent of other options. However, one *logical* option may be stored across

```java
class Locale {
  @StaccatoPropagate(RECEIVER)
  Locale(String country, String lang) { ... }
}
String translateMessage(String msg, Locale l) { ... }
void updateLocale(Locale l) {
  Config.set("country", l.getCountry());
  Config.set("language", l.getLanguage());
}
void writeMessage() {
  String msg = "Hello world!!";
  Locale l = new Locale(Config.get("country"), Config.get("language"));
  output(translateMessage(msg, l));
}
```

Figure 2.7: An example of a single logical option being stored across multiple configuration options. The locale of the program is stored in two options: `"country"` and `"language"`. Notice that the reads of these options in `updateLocale` variable are unsynchronized with the writes in the `updateLocale` method.

several *program* configuration options. This idiom is often used to simulate options that are tuples. For example, in one of our evaluation programs the locale of the program was stored using three configuration options. A simplified sketch of this situation is shown in Figure 2.7.

However, this idiom gives rise to a special kind of consistency error found in the above example. There is no synchronization around the reads of `"country"` and `"language"` on line 12 and the corresponding updates on lines 7 and 8. If the update operation occurs concurrently with the read of the locale options, an invalid `Locale` could be constructed (e.g., the US version of Japanese). Even with the `@StaccatoPropagate` annotation on the `Locale` constructor on line 2, Staccato as described so far would fail to detect this error. In general, if two or more configuration options that represent a logical option are read and updated without coordination, the program may observe an inconsistent version of the logical option.

To detect errors like those described above, Staccato tracks a program's use of logical options in addition to the program options described so far. Staccato maintains a set of *option groups*: each group $\mathcal{G}_i$ is a set of options $o_{(i,1)}, \ldots, o_{(i,n)}$ that together define a logical configuration option named g_i.[3] For example, the program fragment in Figure 2.7 has a single option group: $\{$`"locale"`, `"country"`$\}$. Similar to program options, a logical configuration option g_i is assigned an epoch in the version map, $\mathcal{V}$ and may appear in the domain of configuration histories. The propagation and checking operations also extend naturally to logical options with no changes. Each logical option g_i conceptually maps to a tuple of configuration values: this tuple's elements are the values of the configuration options that make up g_i. Thus, an update of an option $o \in \mathcal{G}_i$ is actually an update of the implicit tuple associated with g_i. Similarly, a read of the option $o \in \mathcal{G}_i$ from a program's configuration abstraction is a projection out of g_i's (implicit) tuple.

This interpretation leads to Staccato's approach to tracking a program's use of log-

[3]The logical option names are internal to Staccato and do not appear in the program text. Staccato generates a concrete name for each option group that does not clash with the program options.

ical options. When an option $o \in \mathcal{G}_i$ is updated, the epoch of g_i in $\mathcal{V}$ is incremented by one. The epoch of o is not incremented. Similarly, the read of an option $o \in \mathcal{G}_i$ from the program's configuration abstraction is tagged with the version of g_i as recorded in $\mathcal{V}$. Thus, all options in a group lose their identities as distinct options.

This simple change is sufficient to detect the found error in Figure 2.7 and others like it. Consider again the scenario where one thread reads the two locale options while another thread updates them. Assume that the option group {`"locale"`, `"country"`} has been given the name g_2, and that initially the $\mathcal{V}[g_2] = 4$. Suppose that the thread executing `writeMessage` executes first, and reads the `"country"` option. According to the rules described above, the value returned from the call to `Config.get` is tagged with the configuration history $\{g_2 \to \langle 4, 0 \rangle\}$. The thread executing `writeMessage` then stalls and control switches to the thread running `updateLocale`, which completes without interruption. During the method, $\mathcal{V}[g_2]$ is incremented once for each call to `Config.set`: at method exit $\mathcal{V}[g_2] = 6$. The value returned from the second call to `Config.get` in `writeMessage` is therefore tagged with $\{g_2 \to \langle 6, 0 \rangle\}$. Thus, after the `Locale` constructor completes, the `l` object will have the configuration history $\{g_2 \to \langle 6, 0 \rangle\} \cup \{g_2 \to \langle 4, 0 \rangle\} = \{g_2 \to \langle 4, 1 \rangle\}$ which is marked inconsistent as desired. Note that any other interleavings of reads and updates of `"language"` and `"country"` will also cause the `l` object to be flagged as inconsistent.

2.4 *Automated Bug Avoidance and Repair*

Staccato also uses the techniques developed for checking to (semi-)automatically avoid DCU errors. Staccato supports two forms of program repair and automatic bug avoidance. The first mechanism (Section 2.4.1) automatically repairs buggy program *schedules* that would expose insufficient synchronization between configuration read and write operations. Staccato detects uses of configuration-derived values and automatically delays any configuration updates that would cause those values to become stale mid-use. Staccato also has a limited form of *value* repair (Section 2.4.2). The programmer may provide a function called by Staccato that repairs stale values when they are dis-

covered. Both mechanisms target reducing violations of the staleness condition. In our experience, this condition is harder to get "right", and therefore benefits the most from automated assistance.

2.4.1 *Coordinating Configuration Reads and Writes*

In a multithreaded environment, it is possible for configuration updates to interfere with an ongoing use of a configuration-derived value. For example, a method may read some initially up-to-date variable x and then later during the same execution re-read x. If an option in x's configuration history is updated between the two reads, Staccato would identify the second read as an error. It is assumed that the program has failed to appropriately react to the configuration update. However, the *use* of x—and by extension the options in x's configuration history—intuitively spans the two read operations. The error occurs because the *update* operation failed to wait for the outstanding use of x (and its corresponding configuration options) to finish. In general, an update of a configuration option being used in another thread without proper synchronization can lead to a DCU error.

Staccato can automatically *prevent* these errors by using a set of per-option read-/write locks, called "option locks". A read acquisition of an option lock indicates that a thread is currently using that option (or some value derived from it). Updates to a configuration option must first acquire the option's lock in write mode. If the option being updated is in use, the write acquisition will stall until all outstanding uses of the option finish and all read holds are released. The option write lock is immediately released after update is effected. The programmer is not responsible for acquiring the write locks during configuration update: all updates are delegated to our implementation's runtime which handles locking (see Section 2.5).

Read-acquisition of option locks is automatically performed by Staccato during a staleness check. Recall that staleness checks occur after every value read. After a value

is read, but before the staleness check is performed, STACCATO read-locks the option locks for every option in the read value's configuration history. These locks are held in read mode after control returns to the host program. The scope of a value's use is approximated as the method containing the initial read of that value. Thus, the read-acquisitions on the option locks acquired during a read are held until the containing method completes. Computing the scope precisely is not possible, so we use end-of-method as heuristic. However, our heuristic was sufficient to prevent several errors that STACCATO would have otherwise detected during our evaluation.

2.4.2 *Value Repair*

STACCATO's value repair is an extension to the staleness checking mechanism. Unlike schedule repair, value repair is not fully automatic: we require the programmer to provide a repair function that updates a stale value to reflect the latest version of the configuration. Although STACCATO can automatically *detect* a stale value, automatically *updating* it to the correct value is beyond pure automation because how to respond in the presence of stale data is fundamentally application-specific.

STACCATO's value repair operates exclusively on object fields. STACCATO already intercepts all field reads for checking. When the STACCATO runtime detects that a value read from a field is stale, it checks if the object hosting the field implements an interface that provides the STACCATO update callback. If the callback exists, STACCATO calls it with the name and current value of the stale field. The callback may rebuild the stale field in an application defined way. The updated field value is then returned from the hook and transparently replaces the old, stale value as the result of the original field read. If the callback is unable to repair a field, the original error is reported as usual; failure to repair a field is not itself an error. Figure 2.8 shows a simplified application of this update mechanism to the example given in Figure 2.1 in Section 2.2.

Repairing (or updating) a stale configuration derived value will likely use the same

```java
class RequestManager implements StaccatoFieldRepair {
  String targetIp, apiKey;
  String doRequest() {
     Request req = new ApiRequest(targetIp);
     req.send(apiKey);
     return req.response();
  }
  Object _StaccatoRepairField(String fieldName,
    Object oldValue, Exception staleException) {
    if(fieldName.equals("apiKey")) {
      return apiKey = Config.get("api-key");
    } else if(fieldName.equals("targetIp")) {
      return targetIp = Config.get("target-ip");
    } else { throw staleException; }
  }
}
```

Figure 2.8: An updated version of the code in Figure 2.1 which uses Staccato repair callbacks. The `Reloadable` interface has been replaced with the `StaccatoFieldRepair`. The new method, `_StaccatoRepairField` is called when Staccato detects that one of the fields is stale. The updated value returned from the method automatically replaces stale field.

options used to construct the old value. As a convenience, Staccato calls the update callbacks while holding the option locks described in Section 2.4.1 in read mode. Thus, although update callbacks may execute in a multithreaded context, the callback effectively sees a consistent, immutable view of the necessary configuration options during the rebuild operation. As a further convenience, Staccato ensures that no threads concurrently execute the update callback on the same object.

2.5 Implementation

We have implemented our technique in a prototype tool named Staccato. Staccato is an offline bytecode instrumentation tool for Java programs. Staccato modifies a program's bytecode to support tracking configuration histories and inserts code to perform checking and propagation. The configuration tracking of Staccato is built on top of a modified version of the Phosphor tool by Bell and Kaiser [17]. Staccato also includes a runtime library that implements the operations described in Section 2.3. Staccato does not automatically integrate with the program's configuration abstraction, this must be performed by the programmer when initially integrating Staccato into the software (see Section 2.5.3).

2.5.1 Basic Operation

Applying Staccato is a two-step process. First, our modified version of Phosphor instruments the source program to add information tracking for primitive types. In a second pass, Staccato uses ASM [111] and Javassist [42] to add code that calls into the Staccato runtime to perform the check and propagation operations described in Section 2.3. For each method covered by a correctness condition, Staccato instruments all array, variable, and field reads, as well as method calls to check read values against the correctness condition selected for the method. Staccato also inserts code at the end of methods annotated with `@StaccatoPropagate` to perform history propagation. Phosphor

already adds unconditional propagation from source values to outputs for primitive instructions such as integer addition; we replaced the merge operation used by Phosphor for these instructions with a call into our runtime library.

Staccato does not require a modified JVM. To ensure information is soundly tracked through calls to the Java Class Library, a program must use a version of the JCL that has been instrumented by Phosphor. To ease integration, we also added propagation to certain "primitive" operations in the JCL, such as string concatenation or string-to-integer parsing. To use Staccato, a program must be launched with a special JVM flag that adds our instrumented versions of the system classes to the system classpath.

2.5.2 *Tracking Configurations*

To reduce memory overhead, Staccato does *not* unconditionally instrument every type to carry configuration history. Staccato uses the programmer's `@StaccatoPropagate` annotations to avoid instrumenting types that will provably never carry configuration information. For a type that *may* carry configuration information, Staccato adds a special field to the class definition. Staccato also modifies the class definition to implement an interface that marks the class as carrying configuration information. This interface exposes two methods that provide the functionality to get and set the hidden field. Staccato synthesizes these methods and inserts them into the class definition. For primitive values (which do not have fields), we reuse the shadow taint variables added by Phosphor.

We chose the interface approach over using reflection for two reasons. First, determining if a type carries configuration information reduces to a relatively inexpensive `instanceof` check. The second reason is to integrate with other dynamic bytecode rewriting tools. There are several libraries that perform instrumentation at runtime to introduce features like database connection pooling.[4] These tools introduce wrapper classes

[4] e.g., http://proxool.sourceforge.net/

that encapsulate the original configuration carrying values. This hiding makes access to a hidden tag field via reflection impossible. However, we found that these tools often make an effort to preserve the interfaces of types being instrumented. This preservation is done at the type level (a wrapper class for some type T also implements the same interfaces as the original type T) and in terms of the behavior of the generated class (by delegating method calls to the wrapped value). Using interfaces allowed Staccato to seamlessly integrate with these dynamic tools. There is no guarantee of interoperability but we found that using interfaces worked with all the dynamic tools we encountered.

2.5.3 Host Program Integration

Staccato does not manage the configuration of the software being analyzed: it is the responsibility of the programmer to integrate the Staccato runtime with the software's configuration abstraction. Staccato assumes that the software uses a key-value data structure for configuration information. The key-value abstraction is widely used in practice for storing configurations, including the Java Properties API, the Windows Registry, and several real-world software projects [158, 96]. All of the software configurations in our evaluation used this abstraction. Rabkin et al. have shown that complex, hierarchical configurations can be adapted to the key-value model [158]. We also assume that all configuration values are strings. In principle, our approach and implementation could be extended to support configuration values of arbitrary types. In practice, all programs we encountered used strings for storage and performed parsing/serialization of these strings where necessary.

Integrating a program's configuration with Staccato requires only minimal source changes. The programmer must delegate all set, get, and delete operations on the configuration key-value store to the Staccato runtime. For example, suppose a program's configuration abstraction is stored in a `HashMap` variable named `conf`. Delegating get operations to Staccato involves changing calls of the form `conf.get(key)` to

`Staccato.get(conf, key)`. Similar changes are made for delete and set operations. The delegation ensures that Staccato increments option epochs on configuration update and correctly tags configuration values read by the program. The API exposed by the Staccato runtime is very general; we were able to incorporate all configuration abstractions we found in our evaluation.

2.5.4 *Propagation Coordination*

Multithreaded execution introduces the possibility for two or more propagation operations to occur concurrently. If two or more propagation operations involve the same object concurrently, the Staccato runtime uses locking to impose an *arbitrary* order on configuration history propagation. The history merge operation (see Sections 2.3.2 and 2.3.4) is commutative: two or more propagation operations with the same target object can occur in any order without changing the final configuration history associated with the object.

Adding locking to arbitrary user programs risks introducing deadlocks, so we modeled the locking protocol of Staccato in the Alloy model finder [95]. Our verification of Staccato's locking was bounded, but we used significantly large parameters to achieve high confidence in our results. Using this model, we verified that the locks introduced by Staccato do not deadlock. Staccato's locks can interact with a program's existing locks to produce deadlocks. In particular, a thread of execution may hold several option locks in read mode, and perform arbitrary lock acquisitions. In practice, this was never a problem in our evaluation.

2.5.5 *Polymorphism and Subtyping*

Staccato does not instrument every read within a method, only those it cannot prove will not carry configuration information. During instrumentation, Staccato may encounter a value of type T that does not itself track configuration information, but there

is some subtype of T that does. In this scenario, STACCATO cannot statically determine if checking should be performed. STACCATO handles this ambiguity by inserting *conditional* checks. If the runtime type of an object does not carry configuration information, a conditional check is a no-op, otherwise a regular check is performed.

Java generics pose a similar problem. Java implements polymorphism using type erasure [29]: `Object` is used to represent type variables at the bytecode level. As every type is a subtype of `Object`, the use of type variable in a target program will lead to ambiguity for STACCATO's instrumentation. However, this situation is simply a degenerate case of the subtyping ambiguity described above. STACCATO therefore resolves all ambiguity caused by generics by exclusively using conditional checks.

2.5.6 Limitations

STACCATO inserts the code that performs the propagation operation at method return after the method body has completed. The propagation target's object's configuration history will therefore be updated *after* the object state has been changed. As a result, the propagation is not necessarily atomic with the body of the method.[5] This can result in the configuration history for an object briefly not reflecting the current state of the object. Unfortunately, it is impossible for STACCATO to automatically incorporate the propagation operation into a synchronization scheme for a method. This limitation did not prevent us from effectively finding bugs and did not admit any false positives.

The association of configuration histories with program values requires that two logical values with distinct configuration histories must have unique identities. This restriction means that STACCATO interacts poorly with memoization or singleton objects, as two or more logically separate configuration histories may be conflated in the same program value. This was primarily a problem for literal strings and enumeration variants. All occurrences of the same string literal and enumeration variant within a single JVM in-

[5] If the method is marked as `synchronized`, the propagation is protected by the object monitor that protects the entire method body.

stance have the same object identity. For literal strings, boolean options were particularly problematic: boolean configuration values were almost always stored using the literals `"true"` and `"false"`. STACCATO conservatively performs a deep-copy when it detects it would otherwise propagate configuration history to a string literal or enumeration.

However, copying enumerations will break a program that relies on the referential equality between two enumeration variants with the same name. To work around this, unboxing operations are added to equality tests that involve enumerations. Unboxing restores a copied enumeration back to the original singleton object for purposes of comparison. This unboxing imposes non-trivial overhead and the programmer must opt-in to this mechanism.

Finally, incorrect annotations by users can produce incorrect analysis results. Over-annotating can lead to false positives. However, these false positives reveal where the programmers assumptions (expressed in his/her annotations) about how a program handles DCU are incorrect. Under-annotation results in bugs being missed, but existing analyses that track the flow of configuration values, e.g. Lotrack [120], and ConfAnalyzer [157], can help guide the user to the correct annotations. This limitation is inherent and would require a user-study to measure the extent of the problem in practice.

2.6 Evaluation

Having defined our approach to finding bugs in dynamic software configuration, we present our evaluation of STACCATO's effectiveness. In evaluating STACCATO, we were interested in the following 4 questions:

1. Does software with dynamic configurability have violations of our correctness conditions?
2. How effective is STACCATO at finding these errors?
3. For some real applications, what is the annotation burden of using STACCATO?

4. What is the performance impact of using STACCATO?

We focused our evaluation on open-source applications with high-levels of run-time configurability. We chose three projects of substantial size but approachable complexity: Openfire (version 3.9.3),[6] a full featured chat server that implements the XMPP IM protocol, JForum (version 2.1.8),[7] a widely deployed forum software, and Subsonic (version 5.2.1),[8] a music streaming server. All projects have many configuration options and extensively use concurrency.

We did not perform an exhaustive evaluation on Solr as the code-base was close to 500,000 SLOC, considerably larger than our next largest evaluation target (Openfire). No technical limitation prevents using STACCATO on Solr: the instrumentation process and dynamic analysis scale independently of code-base size. However, using STACCATO requires understanding a code-base and 500,000 lines felt beyond what we could reliably do ourselves. We did perform an informal evaluation on Solr where we tried to detect the bug mentioned in the introduction using STACCATO. We needed only a handful of propagation annotations and one check annotation to re-find the bug. We believe, but haven't substantiated, that a team familiar with Solr could use STACCATO without undue burden.

Experimental setup We manually annotated each application after first becoming familiar with the code-base and how the software uses configuration values. We also integrated each software's configuration abstraction with the STACCATO runtime.

We were unable to find extensive functional tests for any of the projects. We developed our own functional tests for each of the three software projects. Due to the differences in the software under evaluation, we had to use different evaluation tech-

[6] http://www.igniterealtime.org/projects/openfire/index.jsp

[7] http://jforum.net/

[8] http://www.subsonic.org/pages/index.jsp

niques depending on the software. For Openfire, we used Tsung[9] version 1.4.2. For JForum and Subsonic we used Apache JMeter[10] version 2.12. We developed test plans in these tools that exercised core functionality of each software (these tests are included in the STACCATO distribution). Each functional test client executes in a loop. On each iteration of the loop, a test clients sleeps for a short, randomly selected period of time and then performs a randomly selected test action. These test actions were prepared by us and were designed to use one piece of the core functionality of the software under test. For example, one of the test actions for JForum involves sending a private message from one user to another.

To induce configuration errors, we also developed a *havoc mechanism*. This havoc mechanism consists of several test clients that execute alongside the functional test clients. Each havoc client executes in an infinite loop. During each iteration of the loop, the havoc client sleeps for a short, randomly selected period of time and then performs a mutation to the software's configuration. This mutation was done by simulating an HTTP request to the administrative webpages of the software under test. These administrative webpages also validated our havoc updates (e.g., by rejecting attempts to select negative port numbers). The havoc clients were also restricted to choosing mutations that we had manually prepared in consultation with the program code and documentation. Any errors reported by STACCATO during testing were logged for collection. We report our findings from these experiments in Section 2.6.3.

We evaluated the performance impact of STACCATO along two dimensions, slowdown and memory overhead. To calculate the slowdown, we ran each software's test suite on the instrumented and uninstrumented versions 5 times each and measured response times. Before collecting any data we ran a shorter version of each project's test suite to control for the effects of JIT compilation and application-specific startup actions. We disabled the havoc mechanism during performance testing, as the high rate of configu-

---

[9] http://tsung.erlang-projects.org/

[10] http://jmeter.apache.org/

ration updates (several times per second) is not realistic for measuring performance. To measure the memory overhead of Staccato, we ran the same test suites (again with a brief warmup period) and used Java's JMX technology to monitor the program's memory usage. We sampled the JVM's reported memory usage at one second intervals. Before taking each measurement we triggered a garbage collection.

All experiments were run on a Dell Latitude E-5440 with a 4 core Intel Core i5 processor at 2.00 GHz and with 16GB of RAM. We used version 1.7.0-91 of the OpenJDK JVM.

2.6.1 Summary

The annotation burden of using Staccato is extremely low: the ratio of annotation to SLOC is below 1%. Even including changes to integrate with Staccato and update handlers, we found that the effort needed to use Staccato in a project is minimal. Further, Staccato was able to accommodate most configuration options of interest and use patterns in the projects that we evaluated.

We found DCU errors in all of our evaluation targets, despite widely different approaches to configuration management and multithreading. Many of the DCU errors that we found were violations of the staleness condition caused by two concurrent configuration updates. Two of the projects we evaluated with Staccato also had DCU errors that could occur in a single-threaded context, indicating that DCU errors are not just the result of insufficient testing of multithreaded software. We encountered only one false positive during our analysis.

Staccato imposed a moderate performance penalty in our tests; we measured a maximum overall slowdown of 5.30x and a memory overhead of at most 144.40%. This is overhead low enough to use Staccato as a bug-finding tool in pre-deployment. Our experience suggests combining Staccato with an automated havoc test like those used in our evaluation can be an effective technique for finding DCU errors. Staccato's

Project	Annot.	Flow	Repair CB	Bug-Fixes	Conf-Abs.	Total	SLOC
Openfire	100	184	451	78	212	1,025	85,416
JForum	11	44	13	2	29	99	29,568
Subsonic	13	52	0	0	118	183	29,592

Table 2.1: Counts of lines changed to integrate with STACCATO. **Annot.** counts explicit check and propagation annotations. **Flow** are changes to calling conventions, the introduction of helper methods, or the use of wrapper classes for integration with STACCATO. **Repair CB** are repair callback code. **Bug-Fixes** are fixes for concurrency bugs in the project. These consist primarily of field accesses that were not well ordered according to the Java Memory Model [135]. **Conf-Abs.** are changes to integrate the project's configuration abstraction with the STACCATO runtime. **SLOC** counts the total source lines of code in the original, uninstrumented project.

overhead makes it unlikely that the repair mechanisms described in Section 2.4 can be used in production. However, a developer can use STACCATO's repair mechanisms to rapidly develop DCU implementations or fixes. From an initial implementation using STACCATO, a programmer can develop a more efficient manual solution.

2.6.2 Integration Effort

As a proxy for programmer effort, we measured the number of line changes necessary to integrate STACCATO with each of our evaluation targets. We count explicit propagation and checking annotations as well as lines changed to integrate the software's configuration abstraction with STACCATO's runtime, changes to calling conventions to ease application of annotations, and repair callbacks. The breakdown of lines changed is shown in Table 2.1. The ratio of annotations to total lines for each project is 0.12%, 0.04%, and 0.04% for Openfire, JForum, and Subsonic respectively. Including all source

lines changed gives: 1.20%, 0.33%, and 0.62% for Openfire, JForum, and Subsonic respectively. The relatively high percentage for Openfire can be attributed to large number of lines added for update callbacks (column **Repair CB** in the table).

Qualitatively, most of our effort was spent understanding each code-base. A team familiar with an application would be spared this effort. Adapting a program once we understood how its dynamic configuration worked was largely formulaic. Many bugs were found with no extra annotations as many methods are checked by default against the consistency condition. Finding staleness violations required some manual annotation. However, recall that annotations to control checking by STACCATO can be applied to an entire class. Thus, when we added annotations for finding staleness violations, no pre-existing knowledge of exact bug location was required, only an intuition that a class encapsulated some persistent, configuration-derived state. Finally, we found most candidates for propagation annotations simply by inspecting configuration access sites and determining what methods were called with the newly returned configuration values.

Checking Coverage We found that the configuration model and primitives exposed by STACCATO were sufficient to achieve good coverage of options checked in each evaluation target. Our functional tests exercised between 25%–86% of options in our evaluation programs. STACCATO was able to track and check all options exercised by our tests.

We measured coverage by modifying the STACCATO runtime to record the options checked by STACCATO and the correctness condition being checked. We also recorded which options were read or updated during test execution. Options not involved in at least one update operation are not counted for coverage purposes. Uses of these options were trivially validated by STACCATO but are uninteresting for evaluation purposes. Options that were updated but never re-read *are* included; STACCATO still validated that old copies of the option were used consistently and did not cause violations of the consistency condition.

In all tests, every configuration option mutated by the havoc mechanism was checked

at least once. This result indicates that our approach accurately models the configuration updates of the evaluation programs. Across all projects, most options were checked against the consistency condition rather than the staleness condition. However, all programs used checks for staleness at least once. Openfire and Subsonic both used the condition to find bugs and JForum used staleness checking in combination with repair to add support for configuration updates.

Our tests exercised only a representative set of options for each program in our evaluation. There is no restriction other than annotation and test development effort preventing 100% coverage of options. We felt that achieving full coverage in our tests would provide low marginal benefit for the amount of effort required. For example, we did not test the configuration options that control Openfire's operation in multi-server clusters. An overview of which options we checked during our evaluation is presented in Table 2.2. We calculate the coverage of our tests as:

$$\frac{Checked + Update}{Checked + Update + Untested + Internal}$$

Coverage for our evaluation set is 25.14% 86.36% and 68.75% for Openfire, JForum, and Subsonic respectively. Openfire's relatively low coverage is due to the high number of options available in that program. Some of these options (counted in column Untested) controlled functionality that was difficult to test automatically or that required complex test environments (e.g., multi-server cluster options). We also excluded many undocumented options available in Openfire that cannot be set via normal means (counted in column Internal).

2.6.3 *Effectiveness*

Table 2.3 summarizes the errors that we found in our three evaluation targets. In every evaluation target, Staccato found multiple dynamic configuration update errors. In total, Staccato discovered 20 errors across the three evaluation programs. The most common bugs are staleness violations that occur when two concurrent configuration

Project	Checked	Update	Imm.	Int.	Other	Untested	Total	Cov.
Openfire	24	21	68	73	4	61	251	25.14%
JForum	37	1	3	0	0	6	47	86.36%
Subsonic	33	0	0	0	0	15	48	68.75%

Table 2.2: Classification of options available in each application and coverage of our functional test suite. **Checked** counts options checked during our tests. **Update** are options updated using Staccato's repair mechanism. The next 3 columns count options not included in our tests. **Imm.** are immutable and therefore uninteresting for our evaluation. **Int.** are internal, undocumented options for which we lacked sufficient domain knowledge of the software. **Other** are options that were untestable because of missing proprietary technology or update mechanisms with unfixable concurrency bugs. **Untested** are all remaining options not exercised by our functional tests. **Cov** measures the percentage of options tested in our evaluation.

Project	Stale Read	Incons. Read	Incons. CF
Openfire	11	0	0
JForum	0	5	0
Subsonic	1	2	1

Table 2.3: Counts of errors found by Staccato. **Stale Read** are violations of the staleness condition. **Incons. Read** are violations of the consistency condition. **Incons. CF** are violations of the consistency condition for control-flow.

```java
public void setLogDir(String directory) {
  Config.set("audit.log.dir", directory);
  this.auditDir = directory;
}
```

Figure 2.9: Simplified atomicity bug during configuration update. If two threads concurrently execute `setLogDir`, the value in the `auditDir` field may not be the most recent version of `"audit.log.dir"` option. This violates the object's implicit invariant. Staccato detects this error on subsequent reads of the `auditDir` field.

updates were performed. However, two of the three projects also contain errors that do not depend on concurrency to manifest. We now discuss some example bugs found in each evaluation target.

Openfire

All of the bugs found in Openfire were violations of the staleness condition. The most common errors in Openfire were reads of out-of-date configuration data caused by non-atomic configuration updates. Figure 8 shows a simplified example of the problematic configuration update idiom behind these errors.

We also found a DCU error in Openfire that does not require a specific thread schedule. The server administrator may configure, at runtime, the implementation used to store a user's contact list. Each user's contact list object holds a reference to the storage backend specified by the administrator. Staccato found that contact lists loaded before a change to the storage backend would continue to use the old backend object. This could cause changes made to a user's contact list to be lost. The underlying defect was caused by the contact list objects being cached in a static field. A sketch of this scenario is shown in Figure 2.10.

```
1  static Map<String, ContactList> cache = ...;
2  ContactList getContactList(String user) {
3    if(cache.containsKey(user)) {
4      return cache.get(user);
5    } else {
6      ContactStorage store = getContactStorage();
7      cache.put(user, new ContactList(user, store));
8      return list;
9    }
10 }
11 ContactStorage getContactStorage() {
12   String backend = Config.get("contact.classname");
13   /* reflectively instantiate the backend specified by
14   "contact.classname" */
15 }
```

Figure 2.10: Simplified example of contact list bug in Openfire. A stale version of the backend specified by `"contact.classname"` will persist in ContactStorage objects cached in `cache`.

Using the repair callback mechanism described in Section 2.4, we were able to introduce dynamic updates for 21 options. We also added repair for 2 of the errors that we found in our evaluation. The locking guarantees provided by STACCATO allowed us to write update callbacks that focused only on implementing the actual configuration update and did not require us to add any extra locking. STACCATO also ensured that we did not introduce any new dynamic configuration update errors when adding this new DCU functionality. We validated that our update code was correct by extending the havoc mechanism to include changes to the configuration options for which we wrote update callbacks; no errors were reported by STACCATO in any of the options we tested in this way.

JForum

JForum maintains less configurable global state than Openfire, so STACCATO did not find any violations of the staleness condition. However, STACCATO found 5 violations of the consistency condition.

Most of these errors were the result of two or more reads of the same configuration option without any synchronization. For instance, when generating a response, JForum reads an option twice to set two separate response parameters. The two reads are not protected by any synchronization, and STACCATO detected that a concurrent update of the option would yield an inconsistent response. The single false positive found during our evaluation occurred under similar circumstances. JForum reads a single option twice (again, without synchronization), except that both reads populate the *same* response parameter. STACCATO was unable to detect that the second write overwrote the first, concealing any inconsistency.

One of the consistency errors STACCATO discovered caused JForum to mislabel the encoding used in an HTTP response. Finding this error involved tracking object dependencies at a level not possible with just primitive value dependencies. The bug is caused

by the configuration option that controls response encoding being read twice during request handling: once in the method that generates the HTML response, and in another method that sets the response headers. Using to `@StaccatoPropagate` annotations, Staccato was able to capture the dependencies between the encoding option, the generated output, and the response headers.

Staccato also found a consistency error that does not rely on a specific thread schedule to manifest. JForum pre-computes the URLs of emojis available on the forum and stores the results in a cache. The emoji URLs are built using the configurable location of the forum. However, after the forum location is updated, the emoji URLs in the cache are not updated. The forum location is also read on each request to render the header and footer content of each page. Staccato detected that after the forum URL is updated, responses that include emojis contain two inconsistent versions of the forum URL.

The handful of options that control persistent state all require a restart to take effect, indicating a conservative approach on the part of the JForum developers towards configuration updates. We were able to use Staccato's repair functionality to transparently and safely introduce online updates for one of these options.

Subsonic

Compared to JForum and Openfire, Subsonic has little configurable global state. The one exception was in the LDAP integration component. Subsonic caches its connection to an LDAP server in a static field and rebuilds the connection when it detects that the LDAP configuration has changed. A simplified form of this update algorithm is shown in Figure 2.11. Although the `getConnection` method is synchronized, the method does not synchronize with updates to the software configuration. As a result, `lastChecked` may hold the most recent update time for the configuration but the `ldapConnection` could contain a connection built with an old version of the `"ldap-config"` option. Notice that this update idiom is very similar to the value repair scheme described in Section 2.4.2,

```
1 static LdapConnection ldapConnection = ...;
2 static Time lastChecked = ...;
3 synchronized LdapConnection getConnection() {
4   if(lastChecked < Config.lastUpdateTime()) {
5     ldapConnection = Ldap.connect(Config.get("ldap-config"));
6     lastChecked = Config.lastUpdateTime();
7   }
8   return ldapConnection;
9 }
```

Figure 2.11: A sketch of the Subsonic LDAP connection staleness bug. In this example `Config.lastUpdateTime()` returns the time of the most recent update to the configuration. Although the update code uses `synchronized`, the update still contains an error. If an update to the `"ldap-config"` option occurs between lines 5 and 6, the program incorrectly concludes that the `ldapConnection` field is up-to-date.

although the option locks (Section 2.4.1) prevent a similar error from occurring within our implementation.

Subsonic also contains a control-flow consistency error. A method that controls the music play queue iterates over a list of playlist items and on each iteration checks an option that determines if Subsonic is running behind a firewall. If this option is set, extra processing is performed on the playlist item. Staccato flagged that if the option was changed during an iteration of the loop, Subsonic would inconsistently process the entire list as a result.

One final example bug found by Staccato was caused by one logical option (the server locale) being stored across three different configuration options. The reads of the three options where not synchronized with the update mechanism of the options. As a result, if a read of the three locale options occurred concurrently with an update, Staccato detected that the user would receive an invalid locale (e.g, the US version of Japanese). Detecting this error used the consistency groups described in Section 2.3.4.

2.6.4 Performance Impact

Finally, we measured the performance impact of Staccato on our evaluation targets. We measured two metrics: memory overhead and overall slowdown. Across all evaluation programs, the largest overall slowdown (averaging over test actions) was 5.30x and memory overhead was below 2x. A graph of the slowdown results can be found in Figure 2.12.

We calculated overall slowdown as the geometric mean of the ratio between average response times with and without Staccato, after excluding the largest 5% of response times as outliers. Tsung (the technology used to test Openfire) did not provide individual response times, so we did not filter outliers when calculating Openfire's slowdown. The number and duration of the outliers was broadly similar with and without Staccato, with high variance across runs.

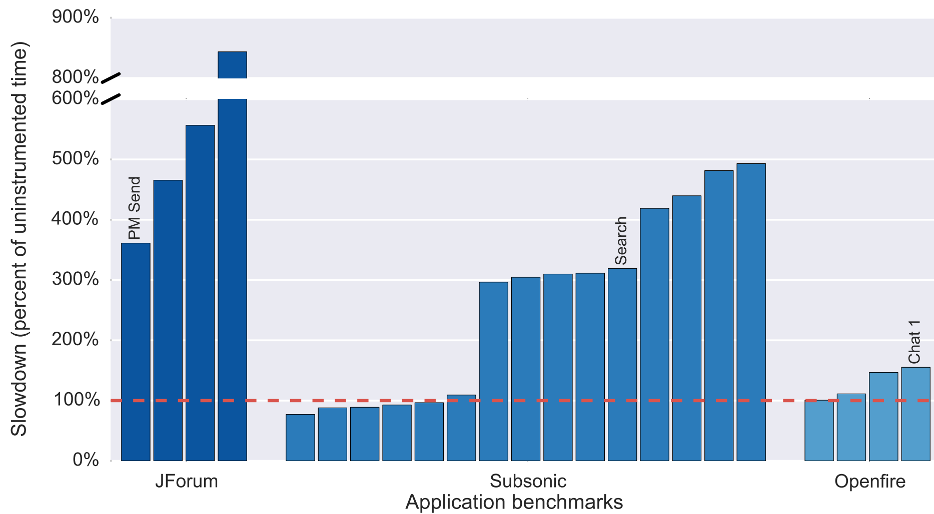


Figure 2.12: The average slowdowns for the evaluation applications. Each individual bar represents the average slowdown of a single test action in the test suites we created for the project. For example, the "PM Send" bar measures the average slowdown experienced when sending a private message to another user on the forum. Similarly, the "Search" and "Chat 1" bars respectively measure the average slowdown for searching Subsonic's music catalog and sending a sequence of chat messages on the Openfire server. The overall slowdown for each application is computed by taking the geometric mean of these average slowdowns.

The overall slowdown for the projects was 1.26x, 5.30x, and 2.11x for Openfire, JForum, and Subsonic respectively. This is generally competitive with other dynamic analyses and is consistent with the performance reported for the Phosphor tool [17] on top of which STACCATO is built.[11] The majority of extra time is spent combining configuration histories for primitive types. History merging is performed via a relatively expensive method call that is inserted after every arithmetic, floating point, and boolean operation. In the common case, the configuration histories being merged are empty. Although we optimized our runtime heavily for this common case, STACCATO encounters the inherent overhead of method calls. STACCATO could be combined with a whole-program static analysis to prune merge operations when the configuration histories involved are provably empty.

The memory overhead of STACCATO was: 110.83%, 144.40%, and 114.43% for Openfire, JForum, and Subsonic respectively. For each application, we calculated the memory overhead by averaging the observed heap sizes during the instrumented test runs, and similarly for the uninstrumented test runs. We took the ratio of these two averages to be the total memory overhead. Most of this memory overhead is added by Phosphor for shadowing primitive variables as noted in the Phosphor paper. Our memory overhead is generally lower than that reported for Phosphor as we do not add tag fields unconditionally to every object as explained in Section 2.5.2. We expect that memory usage could be improved with a static analysis to remove provably empty shadow state.

2.7 *Conclusions*

This chapter presented the first study of errors in dynamic configuration updates. We approached the problem of diagnosing incorrect updates with the dynamic analysis tool STACCATO, which detects stale or inconsistent views of the software configuration. We

[11] The slowdown reported in the Phosphor paper is for a version that supported only integer valued tags. STACCATO is built using a more recent version that supports arbitrary tag types, which is necessarily slower.

evaluated Staccato on three open-source projects and found bugs in all three. Staccato imposes moderate performance overhead, and low annotation overhead. In the following chapter, we complement Staccato with the static analysis Legato which obviates test cases and runtime overhead when finding DCU errors.

Chapter 3

STATIC VERIFICATION OF EXTERNAL RESOURCE CONSISTENCY

3.1 *Introduction*

The dynamic configuration options studied in Chapter 2 are just one piece of the dynamic environment in which today's programs operate. Modern applications use several *external resources*, including configuration files, databases, and network resources. Further, like dynamic configuration options, many of these external resources may change or evolve during program execution. Remote hosts may become unavailable or change their APIs, database entries may be changed by other threads or programs, the filesystem may be changed by other tenants on the program's host. We refer to these changing, evolving resources as *dynamic* external resources; together, these dynamic external resources form an application's view of the dynamic environment in which it executes.

Dynamic external resources generalize the dynamic configuration options presented in Chapter 2. As demonstrated in Section 2.6, handling dynamically changing configuration options is difficult, and this difficulty extends to other types of dynamic external resources. In general, if any dynamic resource is changed between two accesses used in the same computation, a program can observe two or more inconsistent versions of that resource's state, which can lead to arbitrary and often incorrect behavior. Figure 3.1 illustrates contains a program fragment exhibiting a well-known time-of-check-to-time-of-use defect [151, 22, 138] that can lead to a malicious user circumventing filesystem permissions. The attack is possible precisely because the code in question can observe two versions of the filesystem state: specifically, two different versions of the read permission. Such issues are not restricted to filesystems; similar problems can be found in applications that interact with databases with multiple users [13] or that support instan-

```
if(hasReadPermission("harmless_file")) {
    open("harmless_file").read();
}
```

Figure 3.1: Example time-of-check-to-time-of-use bug caused by a dynamic resource update: if `"harmless_file"` is replaced with a symlink to another user's file after the permissions check but before the `open()` call, a leak of another user's private information will occur.

taneous configuration updates as shown previously in Chapter 2.

Further complicating matters, external resources are often mutated by other programs or users of the system without warning and it is often impossible for the application to prevent such changes. For example, in Staccato's DCU setting, configuration updates are frequently performed by one thread concurrently updating the program's configuration without coordinating with other threads. Due to the unpredictability of these changes, errors due to dynamic resource updates are difficult to anticipate, and (like concurrency errors) require difficult-to-write functional tests to manually uncover. Further, although the example shown in Figure 3.1 can be detected with a simple syntactic analysis, the dynamic resource errors we have found in practice often involve multiple levels of indirection through the heap and flows through multiple method calls. There has been extensive work to help programmers contend with and correctly handle these changes [20, 138, 36, 21]. However, existing techniques take a piecemeal approach tailored to a specific resource type (e.g., files [151, 22], our own approach to configuration options in Staccato, etc.).

This chapter presents a unified *static* approach to verify that programs always observe consistent versions of external resource state. This work complements the Staccato analysis with an approach for detecting consistency errors in a *more general* problem area.

Key to our approach is the *at-most-once* condition. The at-most-once condition states that a value may depend on at most one access of each external resource. Intuitively, programs observe inconsistent resource states when a resource changes between two or more related accesses of a resource. By restricting all computations to at most one access per resource, the condition guarantees that every value computed by the program always reflects some consistent snapshot of each resource's state.

We efficiently check this condition for complex, real-world programs using a novel static analysis. Conceptually, our analysis versions the external resources accessed by a program such that each read of a resource is assigned a unique version. Our analysis tracks these versioned values as they flow through the program and reports when two or more distinct versions flow to the same value. Although our analysis focuses on errors caused by concurrent changes, it does *not* explicitly reason about concurrency involving external updates. Our analysis is interprocedural and scales to large programs. The analysis is flow-, field-, and context-sensitive, and can accurately model dynamic dispatch.

We implemented the LEGATO[1] analysis as a prototype tool for Java programs. We evaluated LEGATO on 10 real-world Java applications that use dynamic resources. These applications were non-trivial: one application in our evaluation contains over 10,000 methods. LEGATO found 65 bugs, some of which caused serious errors in these applications. Further, we found that the at-most-once condition is a good fit for real applications that use external resources: violations of the at-most-once condition reported by our analysis often corresponded to bugs in the program. LEGATO had a manageable ratio of true and false positives. Our tool is also efficient: it has moderate memory requirements, and completed in less than one minute for 6 out the of 10 applications in our benchmark suite.

In summary, this chapter makes the following contributions:

- We define the at-most-once condition, a novel condition to ensure consistent usage of external resources (Section 3.2).

[1] LEGATO is open-source, available at https://github.com/uwplse/legato

```
int getDoubled() {
  return Config.get("number") +
    Config.get("number");
}
```

Figure 3.2: Example of inconsistency due to dynamically updated configuration options. If the `"number"` configuration option changes between the two calls to `Config.get()`, a non-even number may be returned.

```
int a = Config.get("number");
int b = 0;
while(⋆) {
  b += a;
}
```

Figure 3.3: Example of a resource used multiple times after being read. $\star$ represents a side-effect free, uninterpreted loop condition. This use pattern is correct because the `"number"` resource is accessed only once in computing `b`.

- We present a novel static analysis for efficiently checking the at-most-once condition (Sections 3.3 and 3.4).
- We describe Legato, an implementation of this analysis for Java programs (Section 3.5).
- We show that Legato can find real bugs in applications that use dynamic resources (Section 3.6).

3.2 *At-Most-Once Problems*

Legato targets programs that use *dynamic external resources*. Unlike static program resources (e.g., program code, constant pools, etc.) dynamic resources are statically identifiable entities that may be changed without warning by code or programs outside of an application's control. In the presence of external changes, programs may observe inconsistent versions of an external resource's state.

For example, Figure 3.2 shows a (contrived) example of an error due to dynamically updated configuration options similar to the one presented in Figure 2.2. Although callers of `getDoubled()` would reasonably expect the function to always produce an even number, an update of the `"number"` option between the two calls to `Config.get()` may result in an odd number being returned. This inconsistency mirrors the unexpected mixing of verbs in the phrase returned by `handleSpotRequest()` in Figure 2.2. This unexpected behavior occurs because the application observes inconsistent versions of the `"number"` option. The time-of-check-to-time-of-use error in Figure 3.1 from the introduction is another example.

One possible technique for *statically* detecting these errors is to concretely model dynamic resource updates and reason explicitly about update/access interleavings. Unfortunately, explicitly modeling concurrency, e.g., [41, 19, 40], is intractably expensive on large programs or requires specific access patterns [121, 124].

LEGATO instead verifies consistent usage of dynamic resources without explicitly reasoning about concurrent updates and reads. In the worst case, a resource may change between *every* access; i.e., every access may yield a unique version of the resource. For example, suppose that the configuration accessed in Figure 3.2 is updated by another thread in response to user input, as was the case in STACCATO's evaluation programs. In the presence of non-deterministic thread scheduling and without prior synchronization between the two accesses of `"number"` on lines 2 and 3, the option may be updated some arbitrary number of times. The current implementation of `getDoubled` correctly handles updates that occur before or after the two accesses: only interleaved updates are problematic.

A key insight of LEGATO is that a program that is correct under the worst-case resource update pattern described above will necessarily be correct under *any* update pattern. Further, under the assumption that every access yields a distinct version of the underlying resource, values from two or more different accesses of the same resource can never be combined without potentially yielding an inconsistent result. It is therefore

sufficient to verify that a value depends on *at most one* access to each resource. Verifying this condition for all values in a program is the *at-most-once problem*.

The at-most-once problem places no restrictions on the number of times a resource may be *used* once read, nor how many times a resource may be accessed, only on how many times the resource may be accessed in computing a single value. For example, the code in Figure 3.3 is correct according to our definition of at-most-once. When the `"number"` option is read on line 1 it reflects a single, consistent version of the option at the time of read. Although `"number"` may be updated an arbitrary number of times as the loop executes, the value of `a` is unaffected by these updates and remains consistent as it is used multiple times during the execution of the loop. As a result, after the loop finishes, the value of `b` will reflect a consistent version of the `"number"` option. If the body of the loop was `b += Config.get("number")`, the at-most-once requirement would be violated.

The at-most-once condition is clearly related to and informed by the consistency condition developed for Staccato. The consistency condition of Staccato is defined over particular dynamic executions of a program, and can admit executions that include multiple correlated accesses to the same resource provided that there are no intervening updates. In other words, Staccato's consistency condition is implicitly stated with respect to a concrete update schedule. In contrast, the at-most-once condition is agnostic to any particular update ordering and must therefore account for all possible update patterns. As a result, it is more conservative and may rule out programs that could be correct under all real-world update patterns.

3.3 *The* **Legato** *Analysis*

Legato is a whole-program dataflow analysis for detecting at-most-once violations in programs that use dynamic resources. For ease of presentation, throughout the rest of this chapter, we assume that there is only one resource of interest that is accessed with the function `get()`. The analysis described here naturally extends pointwise to handle

multi-resource scenarios. Conceptually, the analysis operates by assigning a globally unique, abstract version to the values returned from each resource access. If two or more unique versions flow to the same value, this indicates that a resource was accessed multiple times, thus violating at-most-once.

In a dynamic setting, every read of a resource can be tagged with an automatically incrementing version number. With this approach, detecting violations of at-most-once is straightforward: when two or more different version numbers reach the same value, at-most-once *must* have been violated. This approach is very similar to the Staccato analysis, except in place of updating version numbers in response to resource updates the version numbers are incremented on each access. However, concrete version numbers cannot be translated into the static setting without sacrificing termination or precision.

In place of concrete numbers, resource versions can be *abstractly* represented by the site at which a resource was accessed and the point in the program execution that the resource access occurred. The presence of uninterpreted branch and loop conditions makes it impossible to determine the absolute point in a program execution at which a resource access occurs. Instead, Legato uses *abstract resource versions* (ARVs) to encode accesses *relative* to the current point in the program execution. For example, an ARV can represent "the value returned from the 2nd most recent execution of the statement s", which precisely identifies a single access while remaining agnostic about the *absolute* point in the program execution the access occurred.

The Legato analysis combines a reachability analysis with the abstract domain of ARVs to discover which resource versions flow to a value. The ARV lattice is designed such that the meet of two ARV representing different accesses (and therefore versions) yields $\bot$, which indicates a possible violation of at-most-once.

We first present a simple intraprocedural analysis that does not support loops, heap accesses, or method calls (Section 3.3.2). We then extend the approach to handle loops (Section 3.3.3). The transformers defined by these two sections illustrate the core Legato analysis. In principle, this basic analysis could be extended to extremely conservatively

handle language features, such as the heap or methods. However, in practice, doing so would result in enormous precision loss. We therefore show how we extend the analysis to field- and flow-sensitively handle information flow through the heap (Section 3.3.5). Extending the analysis to precisely handle method calls is non-trivial, and is discussed in Section 3.4. Other program features (e.g., exceptions, arrays, etc.) are straightforward extensions of the ideas presented here.

Abstract Resource Versions As mentioned above, an abstract resource version (ARV) represents a resource version by the access site and the point in time at which the access was performed. To ensure soundness, values returned from different resource accesses must be assigned unique ARVs (we expand on this point further in Section 3.3.4). In a simple language with no loops or methods, ARVs are simply expression labels: each label represents the unique value produced by the execution of the labeled expression. In the presence of loops, we augment these labels with a priming mechanism to differentiate between multiple executions of the same expression. To precisely handle methods, in Section 3.4.1 we generalize to strings of primed labels, which identify an access by the sequence of method calls taken to reach an access site (similar to the approach taken by [193]). Finally, in Section 3.4.2, we further generalize ARVs to sets of strings (represented as tries) to encode multiple possible accesses that may reach a program value. However, even with this representation, our analysis always maintains the invariant that each ARV abstracts a single, unique resource access.

3.3.1 Preliminaries

Before describing the analysis, we first briefly review some relevant background information.

IDE The LEGATO analysis uses the IDE (Interprocedural Distributive Environment) program analysis framework [166]. The IDE framework can efficiently solve program analysis problems stated in terms of *environment transformers*. An environment is a map-

ping from dataflow symbols (e.g., variables) to values. The domain of symbols must be finite, but the domain of values may be infinite, provided the values form a finite-height, complete lattice. The meet of environments is performed pointwise by symbol. IDE analyses assign environment transformers to edges in the program control-flow graph. However, to aid exposition, throughout the remainder of this section we will instead denote *statements* into environment transformers.[2]

The IDE framework targets a specific subclass of analyses where the environment transformers distribute over the meet operator on environments. That is, for all transformers $t : Env \to Env$ and all environments $e_1, e_2, \ldots, e_n$, $\bigsqcap_i t(e_i) = t(\bigsqcap_i e_i)$, where equality on environments is defined pointwise. Given a set of distributive environment transformers, the IDE framework produces a flow and context-sensitive analysis with polynomial time complexity.

Access Paths An *access path* [63, 99] is an abstract description of a heap location. An access path consists of a local variable v, and a (potentially empty) sequence of field names $f.g.h\ldots$ Together, these two elements name the location reachable from v through fields $f, g, h, \ldots$ We will write ϵ to represent an empty sequence of fields, π to refer to an arbitrary (potentially empty) sequence of fields, and $v.\pi$ to denote an arbitrary access path.

3.3.2 *The Basic Analysis*

We first present our analysis on a limited language described by the grammar in Figure 3.4. Every call to `get()` is uniquely labeled with ℓ: we will write concrete labels as 1, 2, etc. Our basic language contains no looping constructs, as a result every `get()` expression is executed at most once. Thus, every access can be uniquely identified by the label of a `get()` expression. For this language, the abstract resource versions are `get()`

[2]This change in presentation does not change the behavior of the analysis; the denotation of a statement is a simplification of the meet of the composition of the edge transformers for all paths through a statement.

(const)	c	::=	`0` | `1` | ...
(var)	v	::=	`a` | `b` | ...
(atom)	a	::=	c | v
(expr)	e	::=	a + a | a | $\texttt{get}^{\ell}\texttt{()}$
(stmt)	s	::=	v = e | s ; s | `skip`
			| `if` a `then` s_1 `else` s_2

Figure 3.4: Grammar for the loop-, and method-free language.

Statement		Transformer
$[\![v_1 = v_2]\!]$	$\triangleq$	$\lambda e.e[v_1 \mapsto e(v_2)]$
$[\![v = c]\!]$	$\triangleq$	$\lambda e.e[v \mapsto \top]$
$[\![v_1 = v_2 + v_3]\!]$	$\triangleq$	$\lambda e.e[v_1 \mapsto e(v_2) \sqcap e(v_3)]$
$[\![v = \texttt{get}^{\ell}\texttt{()}]\!]$	$\triangleq$	$\lambda e.e[v \mapsto \widehat{\ell}]$
$[\![\texttt{skip}]\!]$	$\triangleq$	$\lambda e.e$
$[\![\texttt{if } a \texttt{ then } s_1 \texttt{ else } s_2]\!]$	$\triangleq$	$\lambda e.([\![s_1]\!]e) \sqcap ([\![s_2]\!]e)$
$[\![s_1 \ ; \ s_2]\!]$	$\triangleq$	$\lambda e.[\![s_2]\!]([\![s_1]\!]e)$

Figure 3.5: Environment transformers of the basic analysis.

expression labels: $\widehat{\ell}$ represents the unique version of the resource returned by the corresponding $\texttt{get}^{\ell}\texttt{()}$ expression. Further, at this point the analysis operates on access-paths with no field sequences: we abbreviate $v.\epsilon$ as v.

The basic Legato analysis is expressed using the environment transformers in Figure 3.5. Conceptually, for a program s, the analysis applies the empty environment (i.e., all facts map to $\top$) to the transformer associated with statement s. Thus, the analysis result is given by $[\![s]\!]$ $(\lambda_.\top)$. The analysis is standard in its handling of several language features. For instance, sequential composition of statements is modeled by composing environment transformers, and conditional statements are modeled by taking the meet of the environments yielded from both branches.

The interesting portion of the analysis lies in the handling of variable assignments. Assignments overwrite previous mappings in the environment of the left-hand side with the abstract value of the right-hand side. Integer constants are never derived from resources, and therefore have the abstract value $\top$, which represents any value not derived

from a resource. The statement $v_1 = v_2$ associates v_1 with the abstract version contained in v_2. A resource access of the form $\texttt{get}^{\ell}\texttt{()}$ has the abstract value $\widehat{\ell}$ which, as discussed above, is sufficient to uniquely identify the value returned by the access. This simplified version of the LEGATO analysis is very similar in style to a constant propagation analysis, where in place of integers or booleans, the constants of interest are abstract resource versions.

Values may become inconsistent for two reasons. The first is due to the addition operator. The expression $v_1 + v_2$ is given the abstract value $e(v_1) \sqcap e(v_2)$. The meet operator for these ARVs is derived from a flat lattice:

$$\top \sqcap x = x \qquad x \sqcap \bot = \bot \qquad \widehat{\imath} \sqcap \widehat{\imath} = \widehat{\imath} \qquad \widehat{\imath} \sqcap \widehat{\jmath} = \bot, \text{ if } i \neq j$$

where i and j are two arbitrary labels. If $e(v_1) = \widehat{\imath}$ and $e(v_2) = \widehat{\imath}$, then the result of the addition still depends only on the resource accessed at $\texttt{get}^{i}\texttt{()}$. In this case, at-most-once is not violated, and the meet yields $\widehat{\imath}$ as the abstract value of the addition expression. However, if $e(v_1) = \widehat{\imath}$ and $e(v_2) = \widehat{\jmath}$ then the program is combining two unique versions of the resource, which violates at-most-once. The meet of these two incompatible versions yields $\bot$, which is the "inconsistent" value in the lattice. An example of this behavior is show in Figure 3.6.

Finally, a variable may be assigned $\bot$ due to LEGATO's conservative handling of conditional statements. Recall that the environments produced by the two branches of a `if` statement are met at the control-flow join point of the conditional. Thus, if a variable x is mapped to two distinct, non-$\bot$ values in environments produced by different branches of a conditional, those values will be met yielding $\bot$. In this case, the result of $\bot$ does not correspond to a violation of at-most-once, and is a false positive. The alternative, full path-sensitivity, is unacceptably expensive. We do support a limited form of path-sensitivity to precisely model dynamic dispatch (Section 3.4.2).

```
1 a = 1; // a: ⊤
2 b = get¹(); // b: 1̂
3 c = get²(); // c: 2̂
4 d = a + b; // d: 1̂
5 e = c + d; // e: ⊥
```

Figure 3.6: Results of the basic analysis. The comments on each line show the abstract value assigned to the variable assigned on that line.

```
1 while ⋆ do
2   b = a; // a: 1̂, b: 1̂
3   a = get¹() // a: 1̂, b: 1̂′
4 end
5 c = a + b; // c: ⊥
```

Figure 3.7: Example of priming due to loops. The abstract values shown in comments are derived after executing the loop once. On line 3 Legato primes the abstract value of `b` to distinguish it from the fresh value returned by `get¹()`.

3.3.3 Loops

The simple analysis presented so far is no longer sound if we extend the language with loop statements:

$$\textit{stmt} \;::=\; \ldots \mid \texttt{while}\ a\ \texttt{do}\ s\ \texttt{end}$$

If a $\texttt{get}^{\ell}\texttt{()}$ expression is in a loop, each evaluation of $\texttt{get}^{\ell}\texttt{()}$ must be treated as returning a unique version. However, the transformers presented in the previous section effectively assume $\texttt{get}^{\ell}\texttt{()}$ always returns the same version. We therefore extend the transformers and lattice to distinguish resource accesses from distinct iterations of an enclosing loop.

In a dynamic setting, we could associate every resource access $\texttt{get}^{\ell}\texttt{()}$ with a concrete counter c_{ℓ} incremented on every execution of $\texttt{get}^{\ell}\texttt{()}$. In this (hypothetical) scenario, $\texttt{get}^{\ell}\texttt{()}$ yields the abstract version, $\langle \ell, c_{\ell} \rangle$: by auto-incrementing c_{ℓ} the analysis ensures executions of $\texttt{get}^{\ell}\texttt{()}$ from different iterations are given unique abstract versions.

This straightforward approach fails in the static setting: without *a priori* knowledge about how many times each loop executes, the analysis would fail to terminate. We

introduce *priming* to address this issue. A primed `get()` label $\widehat{\ell}^{n}$ represents the $n+1^{th}$ most recent resource access at $\texttt{get}^{\ell}\texttt{()}$. For example, $\widehat{1}''$ (i.e., $\widehat{1}^{2}$) represents the unique value produced by the third most recent evaluation of $\texttt{get}^{1}\texttt{()}$, and $\widehat{2}$ (i.e., $\widehat{2}^{0}$) is the value returned from the most recent evaluation of $\texttt{get}^{2}\texttt{()}$. Abstract versions with the same base label but differing primes are considered unique from one another in the lattice, i.e., $\widehat{i}^{n} \sqcap \widehat{j}^{m} = \bot \iff i \neq j \vee m \neq n$. Thus, this domain distinguishes between accesses at different `get()` expressions as well as different invocations of the *same* `get()` expression.

The syntactic structure of loops are handled using a standard fixpoint technique. The addition of loops changes how $v = \texttt{get}^{\ell}\texttt{()}$ statements are handled in the analysis. As before, the variable v is assigned the abstract value $\widehat{\ell}$. In addition, a prime is added to all *existing* abstract values with the base label ℓ. We extend the environment transformer for the $v = \texttt{get}^{\ell}\texttt{()}$ case in Figure 3.5 as follows:

$$\lambda e.\lambda v'. \begin{cases} \widehat{\ell} & \text{if } v = v' & (3.1) \\ \widehat{\ell}^{n+1} & \text{if } e(v') = \widehat{\ell}^{n} & (3.2) \\ e(v') & o.w. & (3.3) \end{cases}$$

In other words, the $v = \texttt{get}^{\ell}\texttt{()}$ statement creates a new environment[3] such that:

1. v maps to $\widehat{\ell}$, i.e., the most recent version returned from $\texttt{get}^{\ell}\texttt{()}$
2. Variables besides v that map to the base label ℓ have a prime added, indicating these values originate one more invocation of $\texttt{get}^{\ell}\texttt{()}$ in the past
3. All other variables retain their value from env

A program illustrating this behavior is shown in Figure 3.7.

Termination It is reasonable to wonder if the above environment transformer will add primes forever, i.e., our analysis may not terminate. We can easily show that the analysis

---

[3] Recall that an environment is a mapping of symbols (in this case, variables) to abstract values: the function term $\lambda v'. \cdots$ is such an environment.

will converge by observing that the lattice of primed labels is of finite height (albeit infinite width, as infinitely many primes may be added to a single base label) and then appealing to the termination argument of the IDE algorithm. However, this argument does not provide a good intuition behind why our analysis terminates in practice. We therefore provide an intuitive explanation behind the termination of our analysis.

Let us consider the simple case with a single loop and one call to `get()` labeled ℓ. Variables that are definitely not assigned a value from $\texttt{get}^{\ell}\texttt{()}$ will not be primed and therefore do not affect achieving fixpoint. For variables to which $\texttt{get}^{\ell}\texttt{()}$ *may* flow, the flow occurs along some single chain of assignments, e.g. $\texttt{a = get}^{\ell}\texttt{(); b = a; c = b; ...}$ If instead the assignment occurred along multiple possible chains, the conservative handling of conditionals will yield $\bot$, ensuring the analysis achieves fixpoint.

Consider now the case where some assignments in the chain occur *conditionally*, e.g.:

```
1 while ⋆ do
2   if ⋆ then b = a else skip;
3   a = get¹()
4 end
```

where $\star$ represents uninterpreted loop and branch conditions. In this example, `b` receives the value of some arbitrary previous invocation of $\texttt{get}^{1}\texttt{()}$. Our domain of primed labels cannot precisely represent this value, but the analysis will conservatively derive $\bot$ for `b`, again ensuring the analysis achieves fixpoint. After two iterations of the analysis, two possible values for `b`, $\widehat{1}$ and $\widehat{1}'$, will flow to line 3 from the two branches of the conditional on the previous line. The meet of these two values is $\bot$.

The last case to consider in the single loop case is a chain of definite assignments from $\texttt{get}^{\ell}\texttt{()}$ to some variable ν. For some chain of length k, it is easy to show that the resource will propagate along the chain in at most k analysis iterations. Thus, the resource will flow over the `get()` expression at most k times, and receive at most k primes. After fully propagating along the chain, the value in ν will not receive further primes: on further iterations of the analysis the value in ν is killed by the previous definite assignment in

the chain.

Finally, we consider nested loops. As a representative case, consider the following scenario:

```
1 while ⋆ do
2   b = a;
3   while ⋆ do   a = get¹() end
4 end
```

After the first pass through the outer-loop, the environment produced is $[\texttt{a} \mapsto \widehat{1}]$. On the second pass, the environment that reaches line 3 is $[\texttt{a} \mapsto \widehat{1}, \texttt{b} \mapsto \widehat{1}]$. One further iteration of the inner-loop produces $[\texttt{a} \mapsto \widehat{1}, \texttt{b} \mapsto \widehat{1}']$. The meet of this environment with the previous input environment on line 3 assigns b the value $\perp$, ensuring a fixpoint is reached.

We could have enforced termination by artificially limiting the number of primes on a label to some small constant k. However, we decided against choosing an *a priori* bound for the number of primes lest this bound introduce false positives. However, we found in practice we needed at most 2 primes for the programs in our evaluation set. This finding is consistent with Naik's experience with abstract loop vectors [145], which are similar to our priming approach.

3.3.4 *Soundness*

We have proved that the core analysis presented is sound. We first defined an instrumented concrete semantics that: 1) assigns to each value returned from `get()` a unique, concrete version number, and 2) for each value, collects the set of concrete resource versions used to construct that value. The concrete semantics considers only direct data dependencies when collecting the versions used to construct a given value. We define soundness in relation to these concrete semantics. The Legato analysis is sound if, whenever variable is derived from multiple concrete versions in any execution of the instrumented semantics, the analysis derives $\perp$ for that variable. As our concrete semantics uses only direct dependencies for collecting version numbers, our soundness

claim is only with respect to such dependencies and ignores information propagated via control-flow. We discuss this reasons for this choice further in Section 3.5.5.

Our proof of soundness relies on a *distinctness invariant*: two variables have different abstract resource versions if they have different concrete version numbers under the concrete semantics. In other words, when two variables have the same abstract version, they *must* be derived from the same resource access in all possible program executions. Thus, the invariant ensures that when values derived from different concrete resource versions are combined by a program, the analysis will take the meet of distinct abstract resource versions yielding $\bot$. The converse is also true: if two values with the same abstract resource version are combined, then no program execution will combine two values derived from distinct resource accesses.

The justifications given above for the environment transformers and analysis domain provide intuitive arguments for why this invariant is maintained. The full proofs and concrete semantics are omitted from this chapter for clarity of presentation: they are included in Appendix A. Although our proof is stated only for the simple intraprocedural analysis presented so far, when we extend the analysis to support methods in Section 3.4 we provide an argument for the preservation of the distinctness invariant.

3.3.5 Fields and the Heap

We now consider a language with objects and fields.

$$\begin{aligned} expr &::= \ \ldots \mid \texttt{new}\ T \mid v.f \\ atom &::= \ \ldots \mid \texttt{null} \\ stmt &::= \ \ldots \mid v.f \texttt{ = } a \end{aligned}$$

A subset of the new environment transformers for the heap language are given in Figure 3.8. In this version of the language, our transformers operate on access paths with non-empty field sequences as opposed to plain variables. These environment transformers encode the effect of each statement on the heap: for example, constants, `null`, and `new`

Statement		Transformer
$[\![v.f \;= c \mid \texttt{null}]\!]$	$\triangleq$	$\lambda env.env[\text{pref}(v.f) \mapsto \top]$
$[\![v = \texttt{null} \mid \texttt{new}\ T \mid c]\!]$	$\triangleq$	$\lambda env.env[\textit{pref}(v) \mapsto \top]$
$[\![v_1 = v_2.f]\!]$	$\triangleq$	$\lambda env.env[\textit{pref}(v_1) \mapsto \top, v_1.\pi \mapsto env(v_2.f.\pi)]$
$[\![v_1 = v_2]\!]$	$\triangleq$	$\lambda env[\textit{pref}(v_1) \mapsto \top, v_1.\pi \mapsto env(v_2.\pi)]$

Figure 3.8: New environment transformers for the heap. The *pref*(x) function yield the set of all access paths in e with x as a prefix. In addition, all references π are implicitly universally quantified.

expressions on the right hand side of an assignment "kill" access-paths reachable from the left-hand side.

There are two statement forms that require special care that do not appear in Figure 3.8. First, LEGATO handles assignments with a `get()` right-hand side with the environment transformer from Section 3.3.3 extended to support access paths instead of variables. For an assignment of the form $v = \texttt{get}^{\ell}()$, LEGATO uses the following transformer:[4]

$$\lambda e.\lambda\langle v'.\pi\rangle. \begin{cases} \widehat{\ell} & \text{if } v' = v \\ \widehat{\ell}^{n+1} & \text{if } e(v'.\pi) = \widehat{\ell}^{n} \wedge v' \neq v \\ e(v'.\pi) & \text{o.w.} \end{cases}$$

The second statement form, heap *writes* such as $v_1.f = v_2$, is handled conservatively. LEGATO uses strong updates only for access paths with the $v_1.f$ prefix. After the heap-write, the abstract value reachable from some access path $v_1.f.\pi$ is precisely the value reachable from $v_2.\pi$. However, an access path that only *may* alias with $v_1.f$ is weakly updated. A weak update of the access path $v_3.\pi'$ to the abstract value $\widehat{\ell}^{n}$ takes the

[4]In our formalism, we assume that `get()` returns a primitive value, and thus the environment will only contain mappings for $v.\epsilon$.

meet of the current value of $v_3.\pi'$ with $\widehat{\ell^n}$. Given the definition of the label lattice, this treatment of weak updates means that an access-path $v.\pi$ "updated" via aliasing cannot be updated at all: the new value must exactly match the existing value of the access-path or the access-path may have no value at all, represented by $\top$.

Formally, LEGATO assigns a heap write statement $v_1.f = v_2$ the environment transformer:

$$\lambda e.\lambda\langle v.\pi\rangle.\begin{cases} e(v_2.\pi') & \text{if } v.\pi = v_1.f.\pi' \\ e(v_2.\pi') \sqcap e(v.\pi) & \text{if } v.\pi \neq v_1.f.\pi' \wedge \mathit{mayAlias}(v.\pi, v_1.f.\pi') \\ e(v.\pi) & \mathit{o.w.} \end{cases}$$

Resolving the *mayAlias* query is an orthogonal concern to the LEGATO analysis. In our implementation we use an off-the-shelf, interprocedural, flow- and context-sensitive may alias analysis (Section 3.5.1).

3.4 Interprocedural Analysis

The *inter*procedural version of LEGATO is a non-trivial extension of the *intra*procedural analysis from the previous section. There are two main extensions to the core analysis. First, LEGATO soundly accounts for *transitive* resource accesses. A transitive resource access refers to when a method `m()` returns the result of an invocation of $\texttt{get}^{\ell}\texttt{()}$; the analysis must distinguish between abstract values produced by separate invocations of `m()`. In addition, to analyze realistic Java code, LEGATO precisely models dynamic dispatch. If a method call site `m()` may dynamically dispatch to one of several possible implementations, LEGATO soundly combines the unique abstract values returned by each implementation without sacrificing precision.

Definitions For presentation purposes only, we make the initial simplifying assumption that all methods are static (i.e., all call sites have one unique callee), a method has a single formal parameter `p`, all methods end with a single `return` statement, and method calls are always on the right hand side of an assignment. Later in Section 3.4.2 we show

Transformers for a method invocation: $v_1 = m^k(v_2) \to m(p)\{\ldots; \texttt{return } r\}$		
Call-to-start	**Exit-to-return**	**Call-to-return**
$\lambda e.\lambda\langle v'.\pi\rangle \begin{cases} e(v_2.\pi) & \text{if } v' = p \\ \top & o.w. \end{cases}$	$\lambda e.\lambda\langle v'.\pi\rangle \begin{cases} \rho(e(p.\pi)) & \text{if } v' = v_2 \\ & \wedge \pi \neq \epsilon \\ \rho(e(r.\pi)) & \text{if } v' = v_1 \\ \top & o.w. \end{cases}$	$\lambda e.\lambda\langle v'.\pi\rangle \begin{cases} \top & \text{if } v' = v_1 \\ \top & \text{if } v' = v_2 \\ & \wedge \pi \neq \epsilon \\ \tau(e(v'.\pi)) & o.w. \end{cases}$

Figure 3.9: Interprocedural environment transformers. The names of the columns correspond to the transformer names in the original IDE paper [166]. ρ is a function that transforms values that flow out of a method. τ transforms values propagated over method calls. We will define these methods later in the section. The analysis allows for strong updates to heap locations reachable from the argument of a method, although the base variable retains its value from the caller.

how to support dynamic dispatch, and in practice our implementation supports multiple arguments, multiple return points, etc. We extend the grammar for expressions and statements as follows:

$$expr ::= \ldots \mid m^k(a) \qquad\qquad stmt ::= \ldots \mid \texttt{return } a$$

All method calls are labeled: these sets of labels do not overlap with `get()` expression labels. We will continue to use ℓ to denote an arbitrary `get()` expression label, and k to denote a call site label. We will use the same notation for method call labels used in ARVs (i.e., $\widehat{1}$) as we did for `get()` labels in Section 3.3: context will make clear which type of label we mean.

The interprocedural environment transformers used by Legato are mostly standard in the mapping of dataflow symbols into and out of methods. For a method call $v_1 = m(v_2)$ to $m(p)\{\ldots\}$, the access-path $v_2.\pi$ in the calling context is mapped to $p.\pi$ in the callee method. Dataflow symbols that flow out of a method call (via heap loca-

```
1 m() {
2   while ⋆ do a = get¹() end;
3   return a
4 }
5 b = m²();
6 c = m³()
```

Figure 3.10: A non-trivial interprocedural resource access.

tions reachable from formal arguments, or return statements) are mapped back into the caller environment. Finally, information local to the caller that does not flow through the method call to $\mathtt{m}$ is propagated over the method call.[5] LEGATO's analysis is non-standard only in how values are transformed across method boundaries. Values that flow out of a method are transformed by the function ρ and values propagated over a method call are transformed by τ. We define these functions in this section. The full environment transformers are given in Figure 3.9.

3.4.1 *Transitive Resource Accesses*

In a language with methods, a single primed `get()` label is no longer sufficient to uniquely identify a resource access at some point in time. Consider the code sample in Figure 3.10. After line 5, the value in `b` comes from the most recent invocation of `get`^{1}`()`. However, after `m` is called again on line 6, the value in `b` comes from an execution of `get`^{1}`()` at some arbitrary point in the past. A single-primed label is unable to represent this situation. Leaving the value of $\widehat{1}$ in `b` after the second call to `m` is unsound, and using the $\bot$ value is imprecise. In general, transitive resource access may occur any arbitrary depth in the call-graph.

[5]For readers familiar with the IDE framework, these three components correspond to the call-to-start, exit-to-return-site, and call-to-return-site transformers respectively.

To precisely handle scenarios like the one in Figure 3.10, LEGATO generalizes the primed label ARV into *strings* of such labels. Unlike a single `get()` label, which identifies resource accesses relative to the current point in a programs execution, call-strings encode resource accesses relative to *other* program events: specifically, method invocations. For example, in the above example, the abstract resource version stored in `b` can be precisely identified by "the most recent invocation of `get¹()` that occurred during the most recent invocation of m at call site 2". The call-strings used as ARVs can precisely encode statements of this form.

A call-string takes the form $\widehat{k_1}^p \cdot \widehat{k_2}^q \cdots \widehat{k_m}^r \cdot \widehat{\ell}^n$, where $\widehat{\ell}$ is a primed `get()` label and each $\widehat{k_i}$ is a primed call site label. Call-strings are interpreted recursively; $s \cdot \widehat{k}^n$ represents the $(n+1)^{th}$ most recent invocation of $\texttt{m}^k()$ relative to the program point encoded in the prefix s. The string $s \cdot \widehat{\ell}^n$ has an analogous interpretation. If s is the empty string, the label is interpreted relative to the current point of execution. For example, the resource stored in `b` from Figure 3.10 can be represented by the ARV $\widehat{2} \cdot \widehat{1}$, which has the interpretation given above. As in the intraprocedural analysis, two distinct call-strings encode different invocations of a resource access, and thus their meet returns bottom. The call-strings lattice is a constant, flat lattice, which is a natural generalization of the lattice on individual labels:

$$s_1 \sqcap s_2 = \begin{cases} \ell^n & \text{if } s_1 = s_2 = \ell^n \\ k^n \cdot s_1' \sqcap s_2' & \text{if } s_1 = k^n \cdot s_1' \land s_2 = k^n \cdot s_2' \land s_1' \sqcap s_2' \neq \bot \\ \bot & \text{o.w.} \end{cases}$$

When a value with call-string s flows out of a method m from the invocation $m^k()$, $\widehat{k}$ is prepended onto the string s. In other words, for a method call $v = m^k()$, $\rho \triangleq \lambda s.\widehat{k} \cdot s$. The prepended label encodes that the access represented by s occurs relative to the most recent invocation of m at k. The label $\widehat{k}$ also distinguishes transitive accesses that occurred during the execution $m^k()$ from those resulting from other calls of `m()`.

The intraprocedural fragment of LEGATO remains primarily unchanged. Transitive resource accesses within a loop are handled with a priming mechanism similar to the one used for `get()` expressions. A string with $\widehat{k}$ at the head that is propagated over the method call $v = m^k()$ has a prime added to $\widehat{k}$. We define propagate over method transformer as:

$$\tau \triangleq \lambda s. \begin{cases} \widehat{k}^{n+1} \cdot s' & \text{if } s = \widehat{k}^{n} \cdot s' \\ s & o.w. \end{cases}$$

The justification for this transformation is identical to the one provided for values that flow over `get()` invocations. The added prime indicates any accesses that occurred relative to $\mathtt{m}^k()$ now originate one more invocation of $\mathtt{m}^k()$ in the past. Recursion is treated conservatively but does not require special handling in our analysis. Two iterations of the analysis through a recursive cycle will generate two strings, s and $s \cdot s$, the meet of which is $\bot$, ensuring fixpoint.

Soundness We now informally argue for the soundness of the above approach. Recall from Section 3.3.4 that the soundness of LEGATO relies on a distinctness invariant, which states that if two values are derived from distinct resource accesses LEGATO must assign different abstract resource versions to those values. To simplify the following argument, we will assume that only a single value is returned from the callee via a `return` statement (the argument for values returned via the heap generalizes naturally from the following).

Let us assume the distinctness invariant holds for all values in the caller and callee environments, i.e., values from different invocations of `get()` are assigned different ARVs. Let us then show the invariant holds after the callee returns to the caller. First, it is immediate that the call-to-return transformer τ preserves distinctness for values in the caller environment. Next, suppose the value returned from the callee is derived from some resource access that occurred during the execution of the callee. To preserve the invariant, we must then show that the returned value is given a distinct ARV in the caller. By prepending the label of the call site and priming all ARVs that already contain that label, distinctness is ensured.

3.4.2 Dynamic Dispatch and Path-Sensitivity

Legato is not path-sensitive in general; as mentioned in Section 3.3.2 the abstract value of a variable from multiple branches are met at control-flow join points potentially yielding false positives. A key exception is Legato's handling of dynamic dispatch. In Java and other object-oriented languages, a method call `m()` may dispatch to different implementations depending on the runtime type of the receiver object. In general, it is impossible to predict the precise runtime type of the receiver object for every call site, so a program's static call-graph has edges to every possible implementation $m_1, m_2, \ldots, m_n$ of `m` at the call site $\texttt{m}^k()$. If Legato treated multiple return flows like control-flow constructs such as `if` and `while`, the analysis would be sound but unacceptably imprecise.

Legato handles dynamic dispatch path-sensitively by aggregating results from each distinct concrete callee into a single, non-$\perp$ ARV. Although the resulting ARV encodes multiple, potentially incompatible resource accesses, Legato ensures that all accesses represented by the ARV come from different concrete callees of a single virtual call site. As only one concrete callee is invoked per execution of a virtual call site, only one access represented in an ARV may be realized at runtime. Thus, combining results from different concrete implementations into a single ARV does not allow for violations of at-most-once.

Multiple resource accesses are represented by generalizing the call-string representation from the previous subsection into tries, which encode *sets* of call-strings. Leaf nodes of the trie are labeled with primed `get()` labels, and interior nodes with primed call site labels. The children of a call site node labeled $\widehat{k}^n$ represent the possible results returned from the $(n+1)^{th}$ most recent invocation of the call site with label k. A path through the trie implicitly defines a call-string with the same interpretation as given in Section 3.4.1. The call-string representation of the previous subsection is a degenerate case of the trie representation where each node has only one child.

Formally, we write $\widehat{k}^n \cdot [b_1 \mapsto t_1, b_2 \mapsto t_2, \ldots]$ to represent a call site node $\widehat{k}^n$ with

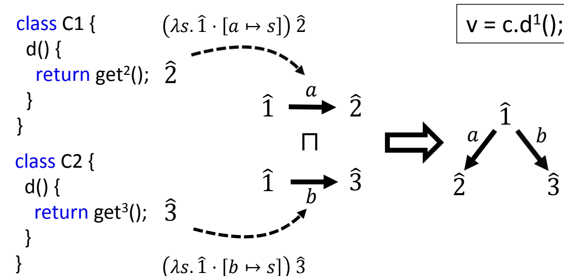


Figure 3.11: Example of LEGATO's handling of dynamic dispatch. `v = c.d¹()` may dispatch to either implementation in `C1` or `C2`. The dashed lines illustrate the return flows, and are annotated with the return flow function applied by the analysis. The two single-child ARVs are met to produce the trie on the right. `a` and `b` are the branch ids assigned to the callees `C1.d` and `C2.d` respectively.

children $t_1, t_2, \ldots$ reachable along branches with ids $b_1, b_2, \ldots$ The branch ids are unique within each call site node and correspond to a potential callee. We call the branch id to child mapping the *branch map*, and write $\mathcal{M}$ to denote an arbitrary mapping.

We extend the return transformer ρ as follows. On return from a concrete implementation m_p to the call site $m^k()$, $\rho \triangleq \lambda s.\widehat{k} \cdot [p \mapsto s]$. That is, the ARV s is extended with a new call site root node labeled $\widehat{k}$ that has a single child with branch id p. In the caller, these single-child ARVs are aggregated into a single node that represents all possible results from each callee. Similarly, we update the function τ as follows:

$$\tau \triangleq \lambda s. \begin{cases} \widehat{k}^{n+1} \cdot \mathcal{M} & \text{if } s = \widehat{k}^{n} \cdot \mathcal{M} \\ s & \text{o.w.} \end{cases}$$

Combining ARVs from *different* invocations of the same virtual call site or different call sites yields $\perp$. To combine ARVs representing results from the *same* invocation of a call site, the branch maps of the ARVs are met pointwise by branch id. As is standard, unmapped branch ids in either map are assumed to have the value $\top$. However, if the

meet of any branch is $\bot$ then the entire meet operator yields $\bot$. That is, a violation of at-most-once in one possible callee yields an overall inconsistent result. An example return flow and meet is shown in Figure 3.11. Formally, the full meet operator for trie ARVs is as follows:

$$\widehat{i}^n \cdot \mathcal{M}_1 \sqcap \widehat{j}^p \cdot \mathcal{M}_2 = \begin{cases} \widehat{i}^n \cdot \mathcal{M}' & \text{if } i = j \wedge n = p \wedge \mathcal{M}' \neq \bot \text{ where } \mathcal{M}' = \mathcal{M}_1 \sqcap \mathcal{M}_2 \\ \bot & \text{o.w.} \end{cases}$$

$$\mathcal{M}_1 \sqcap \mathcal{M}_2 = \begin{cases} \lambda b.\mathcal{M}_1(b) \sqcap \mathcal{M}_2(b) & \forall b' \in \mathrm{dom}(\mathcal{M}_1) \cup \mathrm{dom}(\mathcal{M}_2).\mathcal{M}_1(b') \sqcap \mathcal{M}_2(b') \neq \bot \\ \bot & \text{o.w.} \end{cases}$$

3.4.3 *Effectively Identity Flows*

Prepending labeled nodes on *all* return flows can cause imprecision. For example, consider:

```
1 idA(i) { return i }
2 idB(j) { return j }
3 x = get¹();
4 y = id²(x)
```

where `id` may dispatch to one of `idA` or `idB`. In this example, `x` is assigned $\widehat{1}$ and `y` is assigned $\widehat{2} \cdot [a \mapsto \widehat{1}, b \mapsto \widehat{1}]$. According to the lattice, these two values are distinct and may not be safely combined, despite being identical. This issue arises because the invocation of `id` is unnecessary to identify the resource access that flows to `y`, nor does the behavior of the two possible callees of `id` differ. We call a scenario like the above an *effectively identity flow*.

Legato handles effectively identity flows by detecting when the standard meet operator would produce $\bot$, and refining the ARVs to eliminate any effectively identity flows. Call-site nodes are added on return from a method invocation $m()$ to either identify transitive resources accesses (Section 3.4.1) or to differentiate behavior of multiple callees at

$m()$ (Section 3.4.2). Conversely, if all callees exhibit the same behavior and no transitive resource accesses occur within the call $m()$, call site nodes added on return flow from $m()$ are, by definition, redundant. LEGATO *cannot* add labels on return only when necessary to disambiguate different resource accesses. Such an approach would require non-distributive environment transformers, which are unsuitable for use with the IDE framework upon which LEGATO is built.

Based on this intuitive definition of effectively identity flows, we define a refinement operation $\mathcal{R}$, which traverses the ARV trie, and iteratively removes redundant nodes. After the operation is complete, only the nodes and corresponding labels necessary to either distinguish a resource access or differentiate multiple callees' behavior are left in the trie. We first formally define effectively identity flows (EIF) and initial refinement operation $\mathcal{R}_0$ for the single dispatch case (Section 3.4.3). The definitions of EIFs and the full refinement operation, $\mathcal{R}$, for dynamic dispatch (Section 3.4.3) build upon these definitions.

Effectively Identity Flows and Single Dispatch

As a simplification, we consider call-strings with no primes: the operations and sets defined here can be easily extended to ignore primes on call labels. For every method m, let $\mathcal{AS}(m)$ denote the set of unprimed, call site and `get()` labels transitively reachable from m. Further, for each call site label k we denote the method invoked at k as $\mathcal{CS}_k$. A call-string s contains an EIF if there exists a suffix $\widehat{k} \cdot s'$ such that there exists a $\widehat{j}$ in s' such that $j \notin \mathcal{AS}(\mathcal{CS}_k)$. The existence of $\widehat{j}$ indicates that the ARV must have been returned out of some method other than those called by $\mathcal{CS}_k$, and, by definition, the access represented by the ARV must therefore have occurred in some method other than those called by $\widehat{k}$. Thus, $\widehat{k}$ is irrelevant for the purposes of identifying the resource access encoded in the ARV.

The initial refinement operation, $\mathcal{R}_0$, follows from this definition. Let s be a call-

```
1 id_diff₁(a, b) { return a; }
2 id_diff₂(a, b) { return b; }
3
4 id_same₁(a) { return a; }
5 id_same₂(a) { return a; }
6 v1 = get¹(); v2 = get²();
7
8 r1 = id_diff³(v1, v2);
9 r2 = id_same⁴(v1);
```

Figure 3.12: Effectively identity flows in the presence of dynamic dispatch.

string, $\widehat{k}$ the first label in s involved in an effectively identity flow, and $\widehat{j}$ be defined as above. Finally, let s'' be the suffix of s that starts with $\widehat{j}$ (inclusive). Given these definitions: $\mathcal{R}_0(s) \triangleq \mathcal{R}_0(s'')$. The refinement operation is defined inductively: in the base case where s contains no identity flows the refinement operation is defined to be $\mathcal{R}_0(s) \triangleq s$. Intuitively, the refinement operation iteratively strips off substrings of labels that form effectively identity flows until reaching the suffix of labels that are necessary to distinguish the resource access.

Effectively Identity Flows and Path-Sensitivity

In the presence of ARV branching, we must extend the definition of effectively identity flows presented above. In the single-dispatch case, call site nodes were necessary only to precisely represent transitive accesses; nodes that did not fulfill this purpose could be removed. In the presence of branching, a call site node may also be required to precisely combine otherwise incompatible method call results. Thus, a call site node is part of an effectively identity flow *iff* it is not required to identify accesses within a method call (as before) *and* it does not differentiate two or more otherwise incompatible method results.

For example, consider the code sample in Figure 3.12. The call to `id_diff` may dispatch to either $\texttt{id_diff}_1$ and $\texttt{id_diff}_2$ and similarly `id_same` may dispatch to two possible implementations. After the call to `id_diff` on line 8, `r1` holds the ARV $\widehat{3} \cdot [1 \mapsto \widehat{1}, 2 \mapsto \widehat{2}]$. The two resource accesses $\widehat{1}$ and $\widehat{2}$ do not occur within `id_diff`, but the call site node $\widehat{3}$ is still necessary to differentiate the incompatible results. Therefore, the node $\widehat{3}$ is not involved in an effectively identity flow. In contrast, the value stored in `r2` on line 9 is $\widehat{4} \cdot [1 \mapsto \widehat{1}, 2 \mapsto \widehat{1}]$. In this case, the resource access did not transitively occur within `id_same` and the call site node $\widehat{4}$ is not necessary to differentiate possible results. Thus, $\widehat{4}$ is unnecessary and is part of an effectively identity flow.

We define an effectively identity flow in the presence of branching as follows. Each node encodes a finite set of strings, with each string corresponding to labels on a path from the node to the leaves of the ARV trie. Passing through a call site node $\widehat{i}$ along branch b corresponds to $\widehat{i_b}$. We will denote the set of call-strings for a node n with $n^\natural$:

$$\widehat{\ell}^\natural = \{\widehat{\ell}\}$$
$$\left(\widehat{k} \cdot \mathcal{M}\right)^\natural = \bigcup_{b \in \mathrm{dom}(\mathcal{M})} \{\widehat{k_b} \cdot s \mid s \in \mathcal{M}(b)^\natural\}$$

Similarly a call-string ARV can be trivially converted into a trie ARV as follows:

$$[\![\widehat{i_b} \cdot s]\!] \triangleq \widehat{i} \cdot [b \mapsto [\![s]\!]] \qquad [\![\widehat{i}]\!] \triangleq \widehat{i}$$

Given these definitions, an ARV contains an effectively identity flow if there exists a call site node $n = \widehat{k} \cdot \mathcal{M}$ that satisfies two conditions. First, every call-string $\widehat{k_b} \cdot s \in n^\natural$ contains an effectively identity flow according to the definition in Section 3.4.3 originating at k. In other words, the call site node $\widehat{k}$ is unnecessary to identify any resource accesses within the call at k. The second condition is $\prod_{s \in n^\natural} [\![\mathcal{R}_0(s)]\!] \neq \bot$. That is, after removing the call site node $\widehat{k}$, it must be possible to meet the resulting ARVs without producing a violation of at-most-once. For nodes that satisfy this condition, the full refinement operation is: $\mathcal{R}(n) \triangleq \mathcal{R}(\prod_{s \in n^\natural} [\![\mathcal{R}_0(s)]\!])$. The base case for nodes that cannot

be refined is $\mathcal{R}(n) \triangleq n$. Similarly to the single-dispatch case, the refinement operation traverses the ARV trie, stripping redundant nodes and collapsing redundant branches.

3.4.4 Application Level Concurrency

The at-most-once condition obviates reasoning about concurrent resource updates, but Legato must still account for concurrency within an application. Legato is not sound in the presence of data races: we assume that all mutable, shared state is accessed within a lock protected region. Thus, outside of synchronized regions, each thread reads only values previously written by that thread. However, within a synchronization region, a thread may observe values written by *any* other thread. Legato conservatively assigns heap locations read in synchronization regions the abstract version $\widehat{\ell}$, where ℓ is a fresh, distinct label. In other words, synchronization primitives *havoc* the abstract resource versions potentially shared among threads.

3.5 Implementation and Challenges

We implemented Legato as a prototype tool for Java programs. We used the Soot framework [192] for parsing bytecode and call-graph construction. We built the Legato analysis on an extended version of the Heros framework [25]. Although we state our analysis in terms of access paths for simplicity of presentation, we actually operate on *access graphs* [105] a generalization of access paths. Access paths can only represent heap locations accessible via a finite number of field references. In contrast, access graphs compactly encode a potentially infinite set of paths through the heap. The analysis presented here extends naturally from access paths to access graphs.

To resolve uses of the Java reflection API, we relied on the heuristics present in the underlying Soot framework. However, we also found all of the applications in our evaluation suite provided a mechanisms for one method to invoke another based on an application-specific URL recorded in a static configuration file. We found that, like

many uses of Java reflection [190, 15], these mechanisms are almost always used with static strings. Following the technique outlined in [190], where possible we use these strings to statically resolve these implicit calls to a direct call to a single method. When these heuristics fail, we soundly resolve to all possible callees. Unlike the Java reflection API, which must consider all methods/constructors as possible targets, the set of potential callees was small enough that this over-approximate approach was feasible in practice.

We do not include the full Java Class Library (JCL) in our analysis for performance reasons. For certain methods (e.g., members of the collections framework) we provide highly precise summaries. For unsummarized methods, Legato conservatively propagates information from arguments to return values/receiver objects similar to TaintDroid [69].

3.5.1 Alias Queries

To resolve the *mayAlias* queries on heap writes (see Section 3.3.5), we use a demand-driven, context and flow-sensitive alias resolution [173]. A single alias query must complete within a user-configurable time limit; if this budget is exceeded, Legato reports the configuration value as lost into the heap similar to the approach taken by Torlak and Chandra [186]. This was not a source of any false positives in our evaluation. We take a similar approach on flows of resources into static fields. Static fields are global references that persist throughout the entire lifetime of the program. We conservatively flag any write of a resource derived value that flows into a static field. This dramatically improved our alias resolution time and did not lead to many false positives.

3.5.2 Resource Model

The analysis described in Sections 3.3 and 3.4 is stated in terms of only one external resource. Our implementation handles multiple resources by operating over maps from

resource names to individual ARVs. For generality, our implementation is parameterized over the resource access model of an application. A model defines the resource access sites in an application, and for each site the set of resource names potentially accessed at that site. The soundness and precision of Legato depends on the choice resource model: a model that omits some access sites may cause Legato to miss potential bugs. Similarly, an overly coarse model will be sound but likely imprecise in practice. However, in our evaluation we found that resource access sites are easy to identify in practice; we describe the resource models used in for our evaluation in Section 3.6.

The resource model used with Legato is unconstrained in the choice of resource names. This flexibility allows an imprecise model for when resources may alias, or when the exact name of resources cannot be determined precisely at analysis time. Under an imprecise resource model, all access sites that may access the same concrete external resource are mapped to a common abstract resource name. For example, all accesses to files with the extension `.txt` may be mapped to the logical resource name `*.txt`. A similar approach may be used when two or more resources interact or share state, i.e., resources with distinct names that share state may be given the same abstract resource name.

3.5.3 *Context-Sensitivity*

Each call site of a method m may call the method with different abstract input values. However, the IDE framework computes the values within m by taking the meet over all abstract inputs. This leads to imprecision in the following scenario:

```
1 do_print(a) {
2   print(a);
3 }
4 do_print(get¹());
5 do_print(get²());
```

The standard value computation within `do_print` would assign `a` the value $\widehat{1} \sqcap \widehat{2} = \bot$

which is imprecise. Initial versions of LEGATO used the context-insensitive value computation provided in Heros [25], but our results were impractically imprecise.

To overcome this imprecision, LEGATO extends the value computation phase of Heros to make it context-sensitive. We require an initial context and a context extension operator. At a call site to method m, the context of the call site C is extended with the extension operator, yielding the context C' for values computed within m originating from context C. The original value computation pass of the IDE framework is then executed for the method body with respect to the new context.

In our instantiation, we use an adaptive, k-limited call-string context scheme similar to that in [145]. To trade-off precision and scalability, we initially run the value analysis pass with all contexts limited to length 1. If LEGATO derives the value $\perp$ for some method parameter p in context C, it consults the corresponding argument values in all incoming contexts. If the argument in each incoming contexts is non-$\perp$, LEGATO infers that the $\perp$ value computed for p was due to insufficient context-sensitivity. LEGATO then adaptively increases the context-sensitivity for all such call sites, and then re-runs the value computation phase. This process is repeated until no $\perp$ parameter values arise due to insufficient context-sensitivity, although we impose an configurable artificial maximum length (6 in our experiments) to ensure termination. In our experiments, this limiting was the source of only 3 false positives.

The approach described above is necessarily more expensive than the original IDE framework, which runs only one value computation phase. In practice, the context-sensitive value computation phase does not significantly contribute to analysis time for two reasons. First, LEGATO needs only a handful of value computation phases to either rule out false positives from insufficient context-sensitivity or reach the configured limit. Second, within each value computation phase, values are computed within a method using context-*insensitive* summary functions, which are generated in an initial pass of the IDE analysis. These summary functions are symbolic abstractions of the method behavior on all possible input values. As a result, there is no need to re-analyze a

method under each new context, which keeps recomputing values under new contexts relatively inexpensive.

3.5.4 *Edge Function Representation*

The IDE framework [166] decomposes environment transformers into edge functions between two dataflow symbols. In the LEGATO analysis, the dataflow symbols are access graphs (as discussed above), and edge functions are represented with the trie ARV domain extended with a special parameter node. By using the ARVs tries for both values and function representations, we can reuse the implementations of meet/composition/etc. across the summary function and value computation phases of the IDE algorithm.

The special parameter node, denoted Π, intuitively corresponds to the trie parameter in edge functions. Thus, an ARV consisting of a single Π node corresponds to identity function $\lambda s.s$. Similarly, the ARV $\widehat{k}^i \cdot [b \mapsto \Pi]$ corresponds to the function: $\lambda s.\widehat{k}^i \cdot [b \mapsto s]$. To compose two functions represented as tries $t1$ and $t2$ (denoted $t1 \circ t2$), all instances of Π in $t1$ are simply replaced with copies of $t2$.

We also use the Π node to represent the priming operation. We attach a label/prime pair $(\widehat{k}, n)$ to a Π node to indicate that if the parameter node Π is replaced with a $\widehat{k}$-labeled root then n primes should be added to substituted node. We denote this attachment with $\Pi_{(\widehat{k},n)}$. For example, $\Pi_{(\widehat{1},3)}$ corresponds to the function:

$$\lambda s. \begin{cases} \widehat{1}^{n+3} \cdot \mathcal{M} & s = \widehat{1}^n \cdot \mathcal{M} \\ s & o.w. \end{cases}$$

For concision, we will write $\lambda s.s \oplus (\widehat{1}, 3)$ for functions like the above. We can represent the composition of several such priming functions by allowing a *sequence* of label/prime pairs to a Π node. For example, the composition of the functions $f_1 \circ f_2 = \lambda s.f_1(f_2(s))$

defined by:

$$f_1 \triangleq \lambda s.s \oplus (\widehat{1}, 1) \qquad\qquad f_2 \triangleq \lambda s.s \oplus (\widehat{2}, 2)$$

$$f_1 \circ f_2 \triangleq \lambda s. \begin{cases} \widehat{1}^{n+1} \cdot \mathcal{M} & s = \widehat{1}^n \cdot \mathcal{M} \\ \widehat{2}^{n+2} \cdot \mathcal{M} & s = \widehat{2}^n \cdot \mathcal{M} \\ s & o.w. \end{cases}$$

is represented by the parameter node $\Pi_{(\widehat{1},1)\cdot(\widehat{2},2)}$. In general, we denote a sequence of label/prime pairs by ξ, and will write Π_ξ. A sequence ξ with unique labels has an isomorphism with a total function from labels to natural numbers,[6] and we will use this notation when convenient. The composition of two priming functions represented by Π_ξ and $\Pi'_{\xi'}$, where the labels in ξ and ξ' are disjoint, is simply $\Pi_{\xi\cdot\xi'}$. However, if ξ and ξ' do not contain disjoint labels, e.g., $(\widehat{k}, n) \in \xi$ and $(\widehat{k}, m) \in \xi'$, the resulting function representation must represent a function that adds $m + n$ primes to an ARV with root label $c(k)$. We define the operation $\xi \oplus (\widehat{k}, n)$ as:

$$\lambda\widehat{j}. \begin{cases} n + m & \xi(\widehat{j}) = m \wedge \widehat{j} = \widehat{k} \\ \xi(\widehat{j}) & o.w. \end{cases}$$

And write $\xi \oplus \xi'$, which is the accumulated addition of elements of ξ' via $\oplus$ to ξ. Thus, $\Pi_\xi \circ \Pi_{\xi'}$ is $\Pi_{\xi\oplus\xi'}$. (Note that the concatenation of sequences when ξ and ξ' are disjoint is a special case of $\xi \oplus \xi'$.)

We now define define the substitution operation $\sigma(s, t)$ with priming inductively by:

$$\sigma(\Pi_\xi, \Pi_{\xi'}) = \Pi_{\xi\oplus\xi'}$$
$$\sigma(\Pi_\xi, \widehat{k}^n \cdot \mathcal{M}) = \widehat{k}^{n+\xi(\widehat{k})} \cdot \mathcal{M}$$
$$\sigma(\widehat{k}^n \cdot [b_1 \mapsto s_1, \ldots, b_n \mapsto s_n], t) = \widehat{k}^n \cdot [b_1 \mapsto \sigma(s_1, t), \ldots, b_n \mapsto \sigma(s_n, t)]$$

Thus, the composition $f_s \circ f_t$ of two function f_s and f_t represented by ARV tries s and t respectively is itself represented by the ARV $\sigma(s, t)$.

[6] A sequence ξ that does not contain a label $\widehat{k}$, is functionally equivalent to a sequence $(\widehat{k}, 0) \cdot \xi$. Thus, when converting to a total function, labels that do not appear in ξ implicitly map to 0.

Meet Operations The IDE framework requires a meet operation for the edge function representation. The trivial definition, $f_1 \sqcap f_2 \triangleq \lambda s.f_1(s) \sqcap f_2(s)$, is technically correct but will quickly explode in size. However, as we are using ARVs to represent functions, we can (mostly) use the existing meet operator on ARVs to efficiently implement a meet for functions. The only required extension is to define a meet operator on parameter nodes.

First, consider the simple case of a meet between two priming functions, one of which adds a prime to a root label $\widehat{1}$, the other a prime to label $\widehat{2}$. The naïve meet representation is:

$$\lambda s.(s \oplus (\widehat{1}, 1)) \sqcap (s \oplus (\widehat{2}, 1))$$

Applying this function to the ARV $\widehat{2} \cdot \mathcal{M}$ yields $\widehat{2} \cdot \mathcal{M} \sqcap \widehat{2}' \cdot \mathcal{M} = \bot$. Similarly, the result of applying the resulting function to the ARV $\widehat{1} \cdot \mathcal{M}$ is also $\bot$. Intuitively, the function yields $\bot$ for both example arguments because the two operands of the meet operation add different numbers of primes; i.e., left operand adds 1 prime to a root with label $\widehat{1}$ vs. 0 primes added by the right operand. This intuition hints at the complete definition of the meet operator: the meet of two parameter nodes that add inconsistent numbers of primes for a root label $\widehat{k}$ must yield a parameter node that *returns* $\bot$ *during substitution of a trie with root label* $\widehat{k}$.

Formally, we define the meet operator over prime/label sequences ξ and ξ' as:

$$\xi \sqcap \xi' = \lambda \widehat{j}. \begin{cases} \xi(\widehat{j}) & \xi(\widehat{j}) = \xi'(\widehat{j}) \\ \bot & o.w. \end{cases}$$

Although defined using function notation, the above operation can be efficiently implemented as ξ and ξ' are finite lists of prime/label pairs. The meet operator on parameter nodes is then $\Pi_\xi \sqcap \Pi_{\xi'} = \Pi_{\xi \sqcap \xi'}$. Note that we have extended the domain of label/prime pairs to allow a special "prime" value of $\bot$. Intuitively, if $(\widehat{k}, \bot)$ is attached to a parameter node Π, then when a trie with root label $\widehat{k}$ is substituted for Π the substitution operation should return $\bot$. We therefore extend the substitution operation to:

$$\sigma(\Pi_\xi, \Pi_{\xi'}) = \Pi_{\xi \oplus \xi'}$$

$$\sigma(\Pi_\xi, \widehat{k}^n \cdot \mathcal{M}) = \begin{cases} \bot & \xi(\widehat{k}) = \bot \\ \widehat{k}^{n+\xi(\widehat{k})} \cdot \mathcal{M} & o.w. \end{cases}$$

$$\sigma(\widehat{k}^n \cdot \mathcal{M}, t) = \begin{cases} \widehat{k}^n \cdot \mathcal{M}' & \mathcal{M}' = \sigma_m(\mathcal{M}, t) \wedge \mathcal{M}' \neq \bot \\ \bot & o.w. \end{cases}$$

$$\sigma_m(\mathcal{M}, t) = \begin{cases} \mathcal{M}' & \mathcal{M}' = \lambda b.\sigma(\mathcal{M}[b], t) \wedge \mathrm{dom}(\mathcal{M}) = \mathrm{dom}(\mathcal{M}') \wedge \forall b \in \mathrm{dom}(\mathcal{M}).\mathcal{M}'[b] \neq \bot \\ \bot & o.w. \end{cases}$$

In the above, we extend the $\oplus$ operator on label/prime sequences to handle the special $\bot$ prime such that $x + \bot = \bot + x = \bot$. Additionally, recall that we implicitly lift a partial function from branch ids to ARVs to a total function by using $\top$ for any branch ids without explicit values. We do not explicitly handle this consideration to avoid notational clutter.

Effectively Identity Flows The effectively identity flows discussed in Section 3.4.3 are defined in terms of complete ARVs. This formulation allowed us to exactly determine when a label is required to distinguish transitive accesses or between behaviors of different callees at polymorphic call sites. However, this simple approach is not feasible when taking meets over tries that include the parameter node. One potential solution is to only apply the EIF optimization on ARV's without the parameter node. We found that this was not a realistic approach. We therefore adjust the EIF approach described in Section 3.4.3 to conservatively preserve labels of call sites from which it is possible to reach at least one resource access. In principle, this conservative approach could yield false positives due to preserving call labels in an incomplete trie that are not necessary post-substitution, but we found this conservative approach did not yield any false positives.

3.5.5 Limitations

A fundamental limitation of our analysis is that we do not consider any possible synchronization between resource updates and resource accesses or between multiple resource accesses. This limitation will only yield false positives, as this means our analysis may be overly conservative in considering a program's resource accesses. Our prototype could, with modest effort, include annotations to indicate an access always returns the same abstract version or multiple access sites return the same abstract version.

Our analysis soundness is stated only in terms of direct information flow, i.e., we ignore the effects of implicit flow. Thus, Legato will fail to detect when two or more accesses of the same resource indirectly flow to a program value. We experimented with a version of the analysis that considered implicit flow but, as is common [107], the ratio of false positives to true positives was overwhelming.

As mentioned above, Legato relies on the Soot analysis framework for call-graph construction, reflection resolution, type hierarchy construction, etc. Thus, Legato is sound modulo the soundness of the underlying Soot framework implementation.

3.6 Evaluation

To evaluate Legato, we focused on the issue of consistency in the presence of dynamic configuration updates as studied in Staccato. We chose this problem as representative of the broader problem of consistent dynamic resource usage, as we are unaware of any existing static analysis that is capable of effectively addressing this problem. The only tool we are aware of in this area is our work on Staccato, which may yield false negatives. By instantiating Legato to find DCU defects, we can complement the highly precise Staccato with a tool that (in principle) does not admit false negatives

We are interested in the following questions:

- Does Legato find dynamic resource consistency errors in the analyzed applications with a reasonable ratio of true to false positives?

Program	Classes	Methods	Call Graph Edges	# IR Statements	Options
snipsnap	643	3,318	20,079	68,841	19
vqwiki	506	5,019	43,211	145,891	73
jforum	528	3,075	15,607	41,319	48
subsonic	886	4,578	20,768	67,615	44
mvnforum	938	10,548	132,712	409,847	90
personalblog	371	1,427	8,186	25,514	16
ginp	205	1,011	8,100	26,448	7
pebble	576	2,989	20,646	66,477	7
roller	853	4,735	30,229	95,439	29
blojsom	471	1,782	15,846	26,786	67

Table 3.1: Measures of application complexity in the evaluation suite. **# IR Statements** is the count across all methods of statements in the intermediate representation used by Soot.

- Are the time and memory requirements to run Legato reasonable?

Experimental Setup We evaluated Legato on 10 Java server applications. A summary of the applications and metrics related to code base and call graph size (as measures of application complexity) can be found in Table 3.1. We selected these applications from three sources. Subsonic and JForum come from our prior work on Staccato: we include them for comparison with prior results.[7] Personalblog, Snipsnap, Roller, and Pebble are from the Stanford SecuriBench suite [125], a set of commonly analyzed web apps [126, 118].[8] Finally, we also used applications from prior work by Tripp et al. on TAJ [190], a taint analysis for web applications. We used all projects from TAJ's evaluation that satisfied the following conditions: a) the source code is publicly available, b) the project is a single, self-contained application, and c) the application supports dynamic configuration updates. The applications satisfying these conditions are VQWiki, MVN-Forum, Ginp, and Blojsom. Where possible, we used the same versions of the projects as those used in the original TAJ paper.

The dynamically configurable options of every application may be changed by an administrator at any point while processing a request. All our applications accessed their configurations by reading from a global, in-memory map. When the configuration is changed by an administrator (either via the web interface or editing the on-disk configuration file) a thread in the application updates the in-memory configuration map. This thread runs concurrently with request handler threads that read from the configuration map.

Given this implementation pattern, we treated each individual option as a separate resource that can change at any moment. Every application accessed configuration options by either passing static strings to a key-value API (e.g., `Config.getValue("db-password")`) or calling option-specific getter methods (e.g., `Config.getDBPassword()`). We implemented

[7] Staccato was also applied to Openfire, but it only to detected out-of-date configurations, an orthogonal issue to consistency.

[8] The SecuriBench suite contains 9 applications, but the remaining 5 do not support DCU.

generic resource models for these two access patterns. When analyzing an application, we specialized the appropriate model with an application-specific configuration YAML file which described the application's configuration API. The longest such file was only 195 lines. The number of options tracked for each application are included in Table 3.1.

All of the applications in our evaluation were written to run in a Java Servlet container [141]. To soundly model these applications, we generated driver programs based on the servlet container specification and used sound stub implementations of the servlet API. For heavily used parts of the Java Class Library, such as the collection and database APIs, we used hand written summaries. For other methods without implementations, we used the over-approximation of method behavior discussed in Section 3.5.

We performed two experiments. To measure the effectiveness of Legato, we ran the analysis on each evaluation program, and recorded all at-most-once violations reported by the analysis. We then manually classified these reports as either a true bug or false positive. (Where possible, we reported any true bugs we found to the original developers.)

To measure the performance of Legato, we ran the analysis 5 times for each application while collecting timing and memory usage information. We break down the time of the analysis into three components: call-graph construction time, alias query resolution time, and core analysis time, and report the average of these times. To measure the memory requirements of Legato, we sampled the heap size of the JVM every second. We intentionally avoid garbage collection before sampling the heap size. We found that excessive garbage collection caused an artificially high number of alias query timeouts, which ultimately skewed the analysis results and reported memory requirements.

All experiments were run on AWS EC2 m4.xlarge instances with 4 virtual CPUs at 2.4GHz, using the OpenJDK VM version 1.7.0_131, with 10GB of memory allocated to the JVM. We limited all aliasing queries to ten seconds, and set a 15 minute timeout for each run of the analysis.

Project	TP	FP	PS	SYN	SF	O
jforum	4	14	2	1	3	8
ginp	7	1	0	0	1	0
vqwiki	12	8	2	4	1	1
snipsnap	2	2	1	0	1	0
pebble	0	4	3	0	0	1
subsonic	31	12	1	9	2	0
personalblog	1	3	3	0	0	0
roller	6	5	1	0	1	3
mvnforum	2	27	19	0	0	8
blojsom	t\o	t\o	t\o	t\o	t\o	t\o

Table 3.2: Bug reports from LEGATO. **TP** and **FP** are the numbers of true and false positives respectively. The last four columns record sources of false positives: **PS** is path-insensitivity, **SYN** is the conservative handling of synchronization, and **SF** is the conservative handling of static fields discussed in Section 3.5. **O** counts causes not included in the above categories, and includes imprecision due lack of application-, library-, or framework-knowledge. t\o indicates no reports due to timeout.

3.6.1 Analysis Effectiveness

The results of running Legato on programs in our evaluation suite are shown in Table 3.2. Legato successfully completed within the 15 minute budget on 9 of the 10 applications in our evaluation suite (we discuss the reason for Blojsom's timeout below). Of the 9 applications on which Legato completed, the analysis found bugs in 8. Although the false positive *ratio* is relatively high, we were able to classify the results with minimal effort as many of the false positives were obvious. In many cases (84.6% of column **PS**) Legato detected that it lost precision due to control-flow join and automatically flagged the result as a potential false positive. We also exploited that ARVs are traces of flows from access to report sites to help interpret errors reported by our tool. We were able to find these bugs with a simple resource model (Section 3.5.2) and without being experts in the programs.

There are potentially two sources of false positives: imprecision in the analysis and the at-most-once condition being too strong for application specific reasons. In practice, we found that all false positives were the result of imprecision in the analysis. The primary source of imprecision was the lack of general path-sensitivity in the analysis (column **PS**). For example, almost all of the path-sensitivity false positives in MVN-Forum (16) were the result of identical code being cloned across different branches of conditional statements. The second largest source of false positives was the conservative handling of code that required application-, library-, or framework-specific domain knowledge to precisely model (included in column **O**). For example, 8 false positives in the **O** column of JForum are due to imprecise models of Java's reflection API. Our control-flow graph contained an edge from the return-site of a `Method.invoke` reflective invocation to a `MethodNotFoundException` exception handler, when the represented control-flow path is actually unrealizable.

```
// Instance (1)
request.setAttribute("url", config.getUrl());
request.setAttribute("baseurl", config.getUrl());
// Instance (2)
String url = "/space/" + encode(snip.getName());
url += "/" + encode(att.getName()));
// ...
String encode(String toEncode) {
  String encodedSpace = config.getEncodedSpace();
  return toEncode.replace(" ", encodedSpace);
}
```

Figure 3.13: Two simplified examples of the "double read" pattern found in Snipsnap.

Sample Bugs

We now highlight some of the bugs found and discuss broad patterns we noticed in our results. Many bugs arose from three patterns: 1) two sequential accesses to the same configuration option, 2) using a configuration option in a loop, and 3) storing configuration derived data in a global cache that was not cleared on update.

Double Reads We found 4 instances of applications immediately combining two successive reads of the same option. Two simplified instances we found in the Snipsnap program are shown in Figure 3.13. In the first instance, `config.getUrl()` returns a URL based on the dynamically configurable option specifying the location of the web application. If this option changes between the two accesses, the `request` object's attributes will contain URLs pointing to two different locations. This could cause confusion for the user as only a subset of links on the page returned by Snipsnap would be valid.

The second instance is similar as the two invocations of `encode` both access the dynamically configured `encodedSpace` option. In this instance, the URL returned to the user

```java
List<String> getPodcastUrls() {
  List<String> toReturn = new List<>();
  for(...) {
    String baseUrl = // ...
    int port = config.getStreamPort();
    toReturn.add(rewriteWithPort(baseUrl, port));
  }
  return toReturn;
}
```

Figure 3.14: A correlated access found in Subsonic, where the `"streamPort"` option is aggregated into the `toReturn` variable.

will contain a mix of incorrectly and correctly encoded spaces. As with the first instance, this bug can cause links in the returned page to mysteriously fail to work.

The author of Snipsnap confirmed that these two instances corresponded to true bugs, but declined to fix them due to age of the project, lack of active deployments, and the author's judgment that the bugs were not serious enough to warrant a fix [101].

Correlated Accesses within Loops Out of the 65 true reports, 21 were instances of correlated accesses of configuration options within a loop. We counted instances where a value derived from a configuration option read within a loop is aggregated with configuration-derived values from previous iterations of the same loop. The aggregated value is derived from multiple accesses of the same option, violating our at-most-once condition. The priming approach described in Section 3.3.3 was crucial to detect these bugs.

A simplified example of this pattern, found in Subsonic, is shown in Figure 3.14. The URLs computed by the method are used to generate an XML file served to podcast subscription clients. If some of the URLs generated by the method have inconsistent port

numbers, the subscription client end-user would be presented with a handful of podcasts that fail to work. Further, unlike broken links on a webpage, the generated XML file is likely never seen by the end-user and thus it may not be obvious that a refresh may solve the problem.

We also found this pattern in other applications in our benchmark suite. For example, in MVNForum, a web forum application, the email module may send messages to multiple recipients, but constructs each message in different iterations of a loop. During each loop iteration, MVNForum reads configuration options that specify the message's sender name and address, which may yield a batch of messages with inconsistent sender information.

Finally, we found an example in VQWiki, a wiki web application, that potentially led to a corrupted search index. While constructing the index, VQWiki executes a loop to generate the set of documents to add to the index. Each loop iteration reads a configuration option that controls the location of the application's data files; this value is then stored in the indexed document. If the value of the option were to change between loop iterations, the index would be corrupted and only recover on the next complete index rebuild.

Caching in Static Fields As explained in Section 3.5.1, to avoid expensive alias queries for static fields while retaining soundness, we issue a report for each static field to which resource-derived information flows. This rough heuristic identified 4 instances where the at-most-once condition was violated due to caching.

For example, LEGATO detected JForum's emoji cache bug that was also detected by STACCATO. Recall that JForum computes the URLs for these emojis based on the dynamically configurable location of the forum application. These emoji URLs are cached in a static field, but the contents of this field is not changed if the location of the application is changed. As a result, after the update all links and images will use the new location except for the emojis, which will be broken. Refreshing the page will not fix this issue as it requires the administrator to manually clear the emoji URL cache or restart the

```
1 // in doStartTag
2 this.cols = horiz / Config.getThumbSize();
3 this.rows = vert / Config.getThumbSize();
4 // in doAfterBody
5 if(count - start >= this.rows * this.cols)
6   showPicture = false
7 // in _jspService (autogenerated)
8 int _j_0 = _jspx_getpictures.doStartTag();
9 // 41 lines of auto-generated code
10 int _e = _jspx_getpictures.doAfterBody();
```

Figure 3.15: Inconsistency bug found in Ginp. Detecting this bug requires precisely modeling framework code, and handling flows through method calls and the heap.

application.

In a more serious example, we found an instance in Roller where the login component cached whether password encryption was enabled in a static field populated at startup. However, user administration actions (e.g., update user, create user, etc.) always read the most up-to-date version of this flag, and encrypted passwords as appropriate. Thus, after changing this flag, any new users created by the administrator would be unable to log in until the entire application was restarted.

Other Patterns We found multiple cases where configuration derived values were stored into the heap in one method, and then later combined with a configuration derived value in another method. A minimized example of this pattern, found in Ginp, is shown in Figure 3.15. Like most of the web applications in our evaluation, Ginp uses Java Servlet Pages (JSP), a dialect of HTML which allows mixing arbitrary Java code and user defined tags (such as `<ginp:getpictures.../>`). At page rendering time, JSP pages are translated into Java code and compiled. User defined tags are transformed into a sequence of calls to programmer defined callbacks. However, programmers generally

only interact with the JSP source code and do not see the intermediate code containing these callback invocations.

The bug found by Legato involved one such user-defined tag. In one callback (`doStartTag`, lines 2 and 3), the same configuration option is read twice and stored into two seemingly unrelated heap locations. However, in a second callback (`doAfterBody`, lines 5 and 6) these two values are incorrectly combined to decide a loop condition. Finding this bug required precisely tracing the two abstract resource versions interprocedurally through the heap.

In another example, found in JForum, an SMTP mail session is constructed using the value of the dynamically configured mail host and then stored into an object field. In another method, this session is used to construct a transport, again using the value of the mail host option. If the mail host option changes between these two calls, the transport may try to connect to a mail host different from that of the mail session, which could cause the mail sending process to fail. Further, we confirmed that if the mail sending process failed with an exception, the messages to be sent were dropped and never resent. This bug was also detected by Staccato.

Finally, we found a bug in Subsonic that relied on the sound handling of application level concurrency described in Section 3.4.4. In this instance, a web request would initiate an update of an in-memory list of remote clients. This list was protected by a synchronized block. Legato concluded that a configuration-derived value placed in the list could be mixed with other configuration-derived values that originated from other threads.

Comparison with Staccato To validate the effectiveness of our analysis, we compared the bugs found by Legato with those found by Staccato. A direct comparison is impossible, as Staccato uses a slightly different correctness condition, unsound heuristics not present in Legato, and also detects different types of errors orthogonal to the at-most-once condition. In particular, Staccato detected several consistency violations via the per-method configuration history object described in Section 2.3.4. However, the 4 bugs found by Staccato in JForum and Subsonic that correspond to our at-most-once

```
1 Request req = ...;
2 Response resp = ...;
3 HashMap context = new HashMap();
4 for(Plugin p : plugins) {
5   p.process(req, resp, context);
6 }
7 sendResponse(resp);
```

Figure 3.16: Sketch of code pattern that caused LEGATO to time out while analyzing Blojsom.

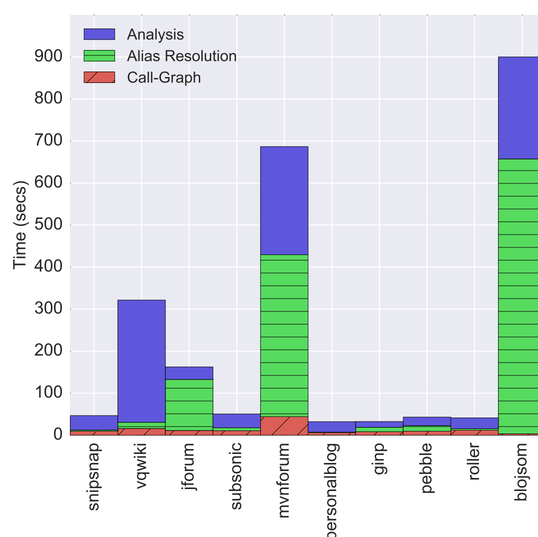


Figure 3.17: Analysis times for the evaluation targets.

condition were detected by LEGATO. This finding partially validates that the bugs found by LEGATO correspond to true DCU bugs.

3.6.2 *Performance*

The results of our performance experiments are shown in Figure 3.17. Of the 10 applications, 9 finished within the 15 minute time limit, and 6 took less than a minute. For all applications in our evaluation suite, the 10GB heap limit was sufficient: the smallest peak heap size we observed was 0.5GB while analyzing Ginp and the largest was 7.5GB on MVNForum.

We now discuss the cause of Blojsom's timeout. The vast majority of Blojsom's 15 minute analysis budget was spent resolving alias queries. We found these expensive alias queries were caused by a problematic code pattern, sketched in Figure 3.16. Blojsom delegates the majority of request processing and application logic to 79 different plugins which are called via interface methods in a for loop during request processing

(lines 4–6). To track per-request state, a shared HashMap `context` is also passed to each plugin; many plugins write configuration information into this map. To find all aliases of `context`, the alias resolver must explore all backwards paths of execution through the loop. Unfortunately, the megamorphic callsite on line 5 caused an explosion in the paths that must be explored, which quickly overwhelmed the alias resolver. We could potentially address this issue by using a less precise approach to aliases, at the cost of overall analysis result quality.

3.7 *Conclusion*

This chapter presented Legato, a novel static analysis for detecting consistency violations in applications that use external resources. Legato verifies the at-most-once condition, which requires that all values depend on at most one access to each external resource. Legato efficiently checks this condition without explicitly modeling concurrency by using abstract resource versions. We demonstrated the effectiveness of this approach on 10 real-world Java applications that utilize dynamically changing configuration options.

Chapter 4

RELATED WORK FOR DYNAMIC EXTERNAL RESOURCES

Linearizability, Consistency, and Atomicity Many of the staleness DCU errors found by Staccato are the result of non-atomic configuration updates. In particular, in Figure 2.9, the program state is not updated atomically with the underlying configuration, which causes a staleness violation in a multithreaded context. In other words, two concurrent invocations of the method in Figure 2.9 are not linearizable with each other [94]. A history of operations on an object is said to non-linearizable if the request/response pairs cannot be reordered into a sequential order that respects the invariant of the object without changing the order of events in the history.[1] Similarly, the consistency conditions checked by Staccato and Legato can be viewed as a linearizability violation. Specifically, a computation that produces a value which reflects one or more versions of a configuration option is not linearizable with one or more configuration updates.

This insight suggests that existing work on linearizability checkers could be applied to the problem of external resource consistency. There has been research into automatically checking linearizability [169, 34, 170] and repairing operations that are not linearizable [124]. However, existing linearizability research focuses on linearizability of ADT operations and cannot handle the arbitrary operations performed on configuration options and other external resources. Further, most linearizability checkers appear to (implicitly) depend on well-defined boundaries for operations. However, our framing of external resource consistency errors as linearizability violations was in terms of arbitrary computations that do *not* have well-defined static boundaries. Thus, we expect applying

[1] In the original linearizability paper, events are given partial order, where $e_1 < e_2$ if the operation of e_1 completes before e_2 begins in the history.

existing linearizability checkers to the problem of external resource consistency will be difficult in practice.

Current work on dynamic atomicity checkers [75, 196] offers another possible solution that could detect consistency violations. In particular, the work in [75] detects violations of serializability among critical sections in code. We can again frame a computation that observes two or more versions of a single resource (configuration options) as within this context. Each update of a resource (configuration option) is given its own critical section, and each computation that reads one or more resources is given its own critical section. If a computation observes multiple versions of a resource then the computation is not serializable with the resource updates. However, the approach of [75] requires that critical sections are again statically known. As demonstrated with the JSP error in Section 3.6 computations that violate consistency can sometimes span multiple (seemingly unrelated) methods making the derivation (automatic or manual) of the correct critical sections difficult in practice.

Finally, we note that existing atomicity and linearizability checkers are best suited to discovering errors in concurrent settings. Although some staleness violations can be framed as linearizability errors, as illustrated by the Openfire contact list example in Section 2.6.3, some staleness violations do not require on concurrency to manifest. It is unlikely that a linearizability or atomicity/serializability checker could effectively detect all staleness violations detected by STACCATO.

Dynamic Software Updates There has been considerable research in the field of dynamic software updates (DSU) [133, 92, 153, 134, 93, 155, 146]. DSU aims to introduce arbitrary code changes to running software without service interruption. Dynamic configuration updates can be viewed as a specific case of DSU: each configuration update is a controlled change to running software at runtime. However, existing DSU research offers a potential alternative for current DCU mechanisms and does not discover bugs in a existing DCU implementations. Further, even in the state of the art, existing DSU tech-

niques require significant effort on the part of the programmer to integrate with existing applications and impose non-trivial performance overhead. One interesting common point between existing DSU research and the approach in Staccato is how to maintain consistent application state in the presence of an online update. Many existing approaches require the programmer to write update hooks (e.g., [146]): these callbacks are similar those used by Staccato for program repair (Section 2.4.2).

Typestate Analysis and Affine Type Systems The "at-most-once" in Legato's at-most-once condition often evokes linear (or more accurately, affine) type systems [195, 84, 188, 28, 60]. Both linear and affine type systems restrict how often a value may be used. Linear type systems guarantee that values may not be duplicated or destroyed, which enforces an exactly-once use discipline. Affine type systems allow destruction, which enforces an at-most-once use discipline. In contrast, under the at-most-once condition resources may be accessed multiple times, and may copied and re-used by the program. The at-most-once restriction only requires that each value depends only on at most one resource access.

Similar to linear and affine types, typestate analyses [175, 61, 205, 59, 74, 144] focus on verifying that the use of some object or resource follows a specific protocol. For example, the motivating example given in the original typestate paper by Strom et al. [175] is verifying that file handles are not written to after being closed. These access protocols are generally expressed in terms of an abstract state assigned to each object, and a set of methods or operations that cause transitions of object state according to some automaton. The at-most-once condition is difficult to accurately capture using this framework. Although it would be possible to design an automaton to enforce that each resource was *used* exactly once during a value's computation, this condition is stricter than Legato's.

Analysis Strategies Staccato's analysis builds on extensive work on dynamic information-flow analysis, in particular taint analysis [83, 90, 17]. Our approach precisely tracks data dependence, but handles handles dependencies from indirect flow using a heuristic. Dynamic analyses that precisely track dependencies from control-flow do exist (e.g., [103, 45]) but suffer scalability that limits their application to large programs such as the ones we use in our evaluation. However, given a scalable framework for control-flow tracking, Staccato could be extended to support control-flow. Legato likewise does not track information propagated via indirect flows for the scalability reasons discussed in Chapter 3.

As mentioned in Section 3.3.2, the Legato analysis is a type of sophisticated inter-procedural constant propagation [39, 198, 106] and builds on a large body of techniques developed in static information-flow analysis. However, constant propagation is generally used within the context of compiler optimizations whereas we are targeting a correctness property.

Configurable Software Configuration management is an active area of research [207, 197, 201, 9, 10, 96, 208, 67, 161, 143, 120, 158, 203]. To our knowledge, no existing research has examined the problem of DCU errors as examined in the Staccato and Legato projects. Some existing work has used approaches similar to the Staccato dynamic analysis to study different problems in the field of configurable software. ConfAid [11] uses a dynamic information-flow analysis similar to ours to diagnose software misconfigurations. It associates each program value with a *configuration set*: this set tracks which configuration options potentially influenced the construction of the value. This is very similar to Staccato's configuration histories. However, ConfAid reasons only about static configurations, and does not track versions like Staccato. Rabkin et al. [157] also tracked configuration values for the purposes of diagnosing configuration errors. However, their analysis does not reason about dynamic configuration updates.

External Resources Many researchers have studied how programs interact with external resources. For example recent work by Linares-Vásquez et al. [123] in the database community generated descriptions of how applications interact with databases. In a related work, Maule et al. among others [156, 136] have evaluated the impact of database changes on applications.

In addition to the above work on static external resources, verifying consistent behavior in the presence of *dynamic*, external resources has also been an active area of research. There has been considerable work in the security field to prevent vulnerabilities due to malicious, concurrent changes of the filesystem [151, 22, 138, 36]. This is similar to check-then-act errors [122, 121, 152] that result from a concurrent update invalidating assumptions established by a check operation.

Several decades of database research on transactions and isolation has focused on ensuring that applications interact consistently with the database. For example, serializable isolation [21] can prevent check-then-act errors within a transaction by determining when a concurrent update has invalidated a previous read. Although this isolation can prevent consistency errors due to concurrent updates, empirical research performed by Bailis et al. [13] has shown that applications that eschew database level transactions (specifically Ruby on Rails applications) struggle to maintain consistency in the presence of concurrent writers.

Chapter 5

WHOLE-PROGRAM STATIC ANALYSIS OF MODERN APPLICATIONS

5.1 Introduction

Static analyses like Legato promise rigorous, automated proofs of the absence of software defects. This promise has slowly gone mainstream: code-analysis tools are now routine for many projects, including at large software companies (such as Microsoft [85, 114], Google [165], and Facebook [38, 37]).

However, building a sophisticated, whole program analysis like Legato is a significant engineering challenge. Luckily, we were able to use existing tools to handle several tedious but necessary tasks during the implementation of Legato. In particular, the Soot framework [191] provided translation from bytecode to an intermediate representation, call-graph construction, type information, string analyses, and points-to information. This functionality is found in a wide variety of analysis frameworks for multiple languages [104, 191, 3, 147, 33]. The developers of these frameworks deserve substantial credit: thanks to these platforms, researchers have been able to ignore complex implementation details and focus solely on implementing their analyses.

Unfortunately, writing a sound static analysis that produces useful results for real programs is still an enormous challenge. Analysis implementations can easily exceed tens of thousands of lines of code [132, 8]; Legato's implementation consists of close to 20,000 source lines of code. To understand the sources of complexity, one need look no further than today's software environment. Industrial-strength analyses must handle industrial-strength applications in industrial-strength languages. Analyses must handle objects, the pervasive use of callbacks, threads, exceptions, frameworks, reflection, na-

tive code, several layers of indirection, metaprogramming, enormous library dependency graphs, etc. In our experience (and those shared by other static analysis authors), getting a realistic static analysis to run on "real" applications requires a combination of luck, multiple heuristics (which may never see the light of day in published papers), engineering effort, manual annotation, and unsatisfying engineering trade-offs. As a concrete example of the luck required, we found during LEGATO development that we needed answers for a type of aliasing query unsupported by all existing off-the-shelf pointer analyses. A few weeks after this discovery, Boomerang [173], which was designed to answer these queries, was published at ECOOP.

The research community has recognized these difficulties and continues to publish work to tackle these challenges. However, developing a novel, sound static analysis *and* testing it accurately on modern applications often remains excruciatingly painful for fundamental reasons. We now describe some of these reasons, using examples drawn from our experience building LEGATO. The difficulties we describe are shared by many other researchers. In particular, in the following section we focus on the challenges posed by huge external libraries, the need for high-level, domain knowledge about API behavior, and the pervasive use of frameworks. These difficulties motivate the second half of this dissertation, which focuses on making whole program static analysis of modern, framework-based applications practical.

5.2 *Static Analysis Challenges*

This section describes the challenges today's static analysis writer faces. Although our descriptions are given in the context of writing the LEGATO analysis for Java, the challenges we identify are not Java-specific in any fundamental way.

5.2.1 *Libraries*

No application is completely self-contained: even a simple "Hello World" application transitively depends on 3,000 classes [112]. The size of an application's transitive dependencies can dwarf the original application code, sometimes by several orders of magnitude. The size of these libraries poses significant scalability challenges for static analysis writers [7]. For example, a highly precise, scalable field-sensitive analysis by Lerch et al. [118] exhausted 25 GB of memory when analyzing the Java Class Library (JCL), which comprises over 18,000 classes. In the same work, an even less precise analysis exhausted the 25 GB memory limit on 6 of 7 non-trivial applications when including external dependencies. Our own experiences broadly mirror this trend: when including all external dependencies Legato exhausts all available memory after running for over 20 minutes.

Some analyses consider all library code along with application code (e.g., [131, 69]). This often limits the sophistication of an analysis: in general the more expressive or complex the analysis, the less scalable it becomes. We do not suggest that useful static analyses that consider library code cannot or do not exist: as mentioned in Section 5.1, large companies run static analyses regularly on their codebases. Nevertheless, considering an application and all dependent libraries requires trade-offs in analysis sophistication and enormous engineering effort.

In practice, the challenges of including all library dependencies means many static analysis writers accept incomplete portions of an application's class hierarchy and/or call-graph. In fact, the Soot framework by default excludes the method bytecode of all classes in the `java` package, i.e., almost the entire JCL. However, ignoring these missing pieces is clearly unsound. Analysis writers therefore resort to one of several unappealing options. The analysis writer may provide hand-written summaries for all missing methods. This approach is precise but infeasible for even moderately sized applications. A significant amount of Legato's implementation effort was spent writing and interpreting summaries of Java's collections library as described in Section 3.5. The collection

library is only a small portion of the entire JCL; supporting the entire JCL would have been a monumental effort.

Another option is to apply a notionally conservative summary of missing library behavior; e.g., "all data flowing into a function are propagated to the return value." This is the approach taken in places by FlowDroid [8] and where LEGATO did not have access to library method implementations and we did not write a manual summary. This technique is still technically unsound as it fails to consider "out parameters" and other side effects, and is also imprecise for pure methods. However, this is often the only option available as even finding the transitive dependencies of all libraries can be impossible.

In response to this difficulty, several authors have explored how to make analysis tractable in the presence of large libraries. A widely explored technique is caching results across runs of an analysis. Caching forms the core of incremental analyses [154, 172, 6, 37, 142, 137]. However, these approaches can reuse results only from previous executions from the analysis on the same program. If an analysis fails to terminate due to large libraries there is no opportunity for caching. Kulkarni et al. have recently proposed a technique to reuse analysis results on common (i.e., library) code shared between two or more target applications [112]. However, this technique can reuse results only of the same analysis and requires programmer provided predicates describing when cached results may be soundly applied. Even the optimal approach for analyzing libraries in isolation remains an area of active research [160].

An alternative option is to write modular (or bottom-up) analyses [52, 89]. Instead of generating summaries for multiple (or infinite) calling contexts in a top-down setting, bottom-up analyses may generate summaries for methods (including library code) valid over *all* calling contexts. However, as noted by Zhang et al. [209], bottom-up approaches may ultimately need to analyze exponentially many input states limiting their scalability in practice. Thus, although theoretically appealing, "designing and implementing [modular analyses] for realistic languages is challenging" [112].

Other authors have investigated generating library summaries in isolation for use in client analyses. A widely cited technique by Rountev et al. [164] can generate some summaries in isolation, but places limitations on the library call-graph and appears unlikely to scale for analyses with a rich domain of input facts. In particular, their technique enumerates over all possible inputs to a method. Even for relatively simple domains this approach is unlikely to scale.

Finally, instead of relying on the hand-written or unsound rule-of-thumb summaries described above, many authors have explored automatically inferring specifications for missing library methods [24, 58, 127, 149, 159]. For example, in the context of a taint analysis, Bastani et al. [16] infer the specifications for missing methods needed to complete flows from sources to sinks. These specifications are presented to the user as candidate method specifications. Albarghouthi et al. and Zhu et al. [5, 211] have both explored using abduction to infer the minimal method specifications to verify the absence of errors. These techniques are promising, but they are currently limited to relatively simple specifications, require a human oracle, or focus on inferring preconditions for methods. These limitations mean that these techniques are unlikely to infer, e.g., the behavior of Java's thread pool or executor APIs.

The decision to exclude library implementations is motivated by scalability concerns but also affects soundness. What impact do these decisions have on the analysis results reported in the literature? It is hard to say: the answer is certainly "a non-zero number" but to our knowledge there is no empirical study on false negatives due to excluded library code nor is this commonly reported in existing analysis results. It is up to analysis evaluators (who are usually also the analysis designers and implementers) to decide if this unsoundness arises in practice for the applications being analyzed. Unless the community can devise convincing experiments that the effects of excluding library code are negligible, the current approaches used may undermine the credibility of static analysis results.

5.2.2 *High-Level API Knowledge*

Analysis writers often require domain knowledge about the behavior of an API. For example, to soundly construct call graphs, analyses must handle the concurrency and reflection APIs of the Java Class Library. The reflection and concurrency APIs are just one example: many different analyses need high-level knowledge about an API. For example:

- What methods read or write from the database? [181]
- What methods return personal or sensitive information? [8]
- What methods may block execution of the current thread? [110]
- What methods and classes are part of a container abstraction? [65]

The answers to these questions are difficult to extract automatically and require reading the relevant documentation. Unfortunately, a static analysis developer interested in the answers to these questions must manually audit an API to find the methods of interest. This audit is no trivial task: the reflection API alone contains over one hundred methods spread across 17 different interfaces and classes. The identified methods are added to a list of "special" methods; the analysis developer must then incorporate *ad hoc* handling for these methods to the analysis. For example, the call-graph construction facility of Soot [191] contains a hard-coded list of reflection and thread methods. WALA [3], another framework for Java analysis, maintains its own list in an external XML file.

For many combinations of analysis domains and APIs, it is likely another analysis author has already performed a similar audit. However, no shared infrastructure exists to reuse and share the results of these audits, condemning analysis writers to re-audit APIs. In addition to wasting time, this process is error prone: failure to properly account for high-level API knowledge may make an analysis unsound. Boomerang [173],

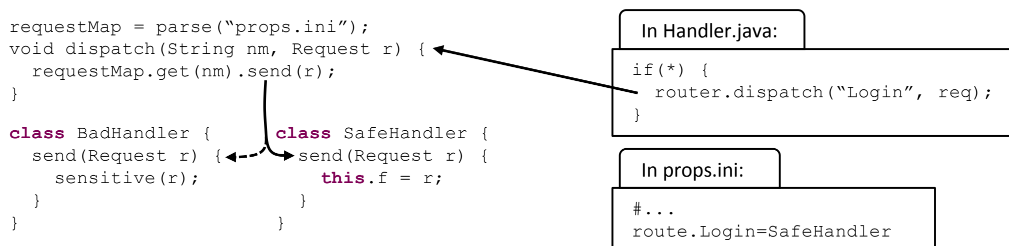


Figure 5.1: An invented program fragment that demonstrates string indirection commonly found in frameworks. Without the routing information in `props.ini`, the analysis must conservatively assume the program dispatches to `BadHandler` (dashed line). Many framework models incorporate this type of information.

an otherwise sound and precise alias analysis, failed to consider `Class.newInstance()` an allocation site and therefore could not find aliases of reflectively instantiated objects.

5.2.3 *Frameworks*

Applications in complex domains (e.g., web applications, GUI programs) require a common set of functionality that does not vary significantly from application to application. For example, most web applications must parse incoming HTTP requests and dispatch them to the appropriate handler code. Rather than reimplement this functionality, applications use *frameworks*.[1] Frameworks are skeleton applications with holes for application specific code. Frameworks generally handle "boring" tasks (e.g., parsing HTTP requests or dispatching incoming UI events) and allow the programmer to focus on application specific tasks, e.g., responding to an HTTP request or UI event.

Frameworks are notoriously hard to analyze. In the interest of reusability, framework implementations rely heavily on language features that are difficult or impossible to analyze in general, such as reflection [23, 128]. This design makes basic call-graph

[1]The line between a library and framework is fuzzy. In this context, we use framework to refer to code that provides scaffolding upon which an application is built.

construction (a basic requirement of any whole program static analysis) incredibly difficult. In addition to reflection, frameworks often use multiple layers of abstraction that confound most static analyses. For example, in Figure 5.1, finding the exact callee of `send()` in the `dispatch()` method requires reasoning about the precise key/value pairs present in the `requestMap` variable. Without this information, the static analysis must conservatively assume any handler is invoked, leading to a false report in `BadHandler`.

Making matters even worse, frameworks are often configured using annotations [76], XML files [2], or other static sources. For example, the mapping information necessary to precisely resolve the `send()` call in Figure 5.1 is found only in the configuration file `props.ini`. A more extreme example of configurations, simplified from an application we encountered while evaluating LEGATO, is shown in Figure 5.2. This configuration snippet controls the (reflective) construction of an object graph, and contains a filtering DSL that is interpreted by one of the reflectively instantiated objects. In this example, the behavior of both the object construction facility and the DSL interpreter are almost entirely controlled by this configuration information. A static analysis must either consider these external artifacts (which requires deep domain knowledge) or make conservative assumptions about the behavior of the framework (leading to a precision loss and corresponding performance hit). Ignoring a framework's code entirely is not a realistic option: applications written using frameworks often lack a distinguished "main" function making even basic call-graph construction impossible.

In practice, static analysis writers either laboriously manually write models[2] of frameworks [189, 8] or avoid evaluating their analysis on framework applications. The latter option is unrealistic considering trends in application engineering but is understandable given the former option: constructing framework models by hand is a time-intensive and frustrating process. Our own experience evaluating LEGATO on web applications that use the Servlet framework is representative of this difficulty. The Servlet frame-

[2]A model is a compact, potentially non-executable, description of framework behavior.

work is relatively simple but building a sound model of the framework required reading parts of three specification documents: the Servlet, JSP (JavaServer Pages) and EL (Expression Language) specifications, which together total 557 pages of prose. The Servlet framework is not an outlier: the reference document for Spring [2], a framework used by Subsonic that builds on the Servlet framework, totals *910* pages.

Unfortunately, a good framework model usually cannot be built based on the framework documentation itself. As mentioned above, framework behavior is often heavily influenced by the application-specific configuration information. The analysis author must often write *model generators*, which interpret a program's framework configuration file and produce a sound model of the framework's behavior for that specific application and configuration. This generated model is often a program that simulates (in an often *ad hoc* way) the framework's execution. For example, the Android framework dispatches UI events to Android activities and can switch between activities based on lifecycle events. The FlowDroid project models this behavior by generating a dummy main method which contains a nondeterministic event loop that calls into the application's event handlers. Legato took a very similar approach; we wrote a model generator that generates a main method which simulates receiving HTTP requests and the appropriate handlers. The dummy main method generated by Legato also modeled the invocation of startup listeners, filter chains and request listeners. Further, some applications in our evaluation used the Struts or Spring frameworks that themselves build on the Servlet framework. We also built model generators for these frameworks, itself a significant challenge.[3]

Building a good model requires more than just understanding the framework and building a sound model. A model must also be precise enough that client analyses can complete in reasonable amounts of time. For example, the largest performance gains in Legato did not come from optimizations in the core analysis, but from aggressively including more application-specific configuration information into our Servlet model to

[3] We released these model generators as the standalone `simple-servlet` project on GitHub: https://github.com/uwplse/simple-servlet.

```
1 <bean id="filterChain" class="FilterChain">
2   <property name="chain">
3       PATTERN_TYPE_APACHE_ANT
4       /logout=logoutFilter,anonymousProcessingFilter
5       /login=basicProcessingFilter,rememberMeProcessingFilter
6   </property>
7 </bean>
```

Figure 5.2: A simplified framework configuration fragment. The `filterChain` "bean" is bound to an instance of `FilterChain`. The values of fields of the `chain` bean are configured with property elements (line 2). Lines 3–5 define a tiny URL-mapping DSL stored in the `chain` field. Building a complete model of this configuration requires not only a model of bean definitions, but an interpreter for this DSL.

improve call-graph precision. These improvements are often non-trivial to implement and require far-reaching modifications to the core analysis and libraries.

For example, the Servlet framework includes a forwarding mechanism similar to that shown in Figure 5.1. The forwarding API takes a URL and returns a *dispatcher object* which itself exposes an API that forwards the request to the handler associated with the URL. This functionality is illustrated on lines 2 and 3 in Figure 5.3. As with most framework behavior, the handler invoked by the dispatcher object is determined by a mapping in the application's configuration file. Our initial, naïve model returned a dispatcher that forwarded to all possible handlers. This approach led to enormous cycles in the call-graph, overwhelming the alias analysis we used in Legato. To improve precision we implemented the following optimization. For each handler h defined in the Servlet's configuration file, we generated a dispatcher object class h^c that unconditionally dispatches to h. The dispatcher object for the `FooHandler` is illustrated on lines 6–11 of Figure 5.3. Wherever the program invokes the forwarding API with a URL u that

```
1  // pre instrumentation
2  RequestDispatcher disp = serv.getDispatcher("/foo");
3  disp.forward(req, resp);
4
5  // Generated by Legato
6  class FooHandlerDisp implements RequestDispatcher {
7    void forward(Request req, Response resp) {
8      // unconditionally forward req and resp to the FooHandler
9      fooHandler.handle(req, resp);
10   }
11 }
12
13 // post instrumentation
14 RequestDispatcher disp = (FooHandlerDisp)serv.getDispatcher("/foo");
15 disp.forward(req, resp); // resolves to FooHandlerDisp above
```

Figure 5.3: An example optimization used during analysis of Servlet-based applications. The instrumentation forces the call-graph construction algorithm to resolve the `forward()` call to a special implementation that invokes the appropriate handler.

is handled by h, we downcast the result to h^c. This instrumentation is illustrated on lines 14 and 15 of Figure 5.3. This downcast forces Soot to resolve any `forward()` calls to the unconditional dispatcher implementation we generate. Although this optimization proved effective and significantly improved call graph quality, it required modifying Soot's call-graph construction facility, our model generation, and the core LEGATO analysis. Note also that these modifications were part of just one of several optimizations we implemented during the LEGATO project.

Given the immense engineering effort, time, and careful planning required, the research community recognizes the difficulty of building a framework model: as recently as 2015 [23], a complete model of the Android framework was a significant research contribution. The work cited in [23] is just one of several efforts tackling analysis of applications built on the Android framework [8, 132, 79, 204, 210]. Unfortunately, the resulting models for frameworks (Android and others) are not always reusable across frameworks, analyses, or application domains. This lack of generality would not be significant if there were only a small handful of frameworks in use today; researchers could simply devote the required time to create good models for this small set of frameworks. However, there are *many* frameworks in active development. Although, there has been work to simplify writing these models using a DSL [174], expecting static analysis writers to build sound and efficient models for every framework in existence is unrealistic. However, evaluating new analyses on applications that use older, simple-to-model frameworks is equally undesirable as it ignores trends in modern software development.

5.3 *Conclusion*

Despite advances in tooling and mainstream success, static analysis development is still a painful process. The difficulties described above motivate the second part of this dissertation, which shows how to make the analysis of modern, framework-based applications tractable. The work described in the following two chapters does not address all of the challenges described above. Some potential solutions for the remaining challenges are

presented as future work in Chapter 9.

Chapter 6

HYBRID MOSTLY-CONCRETE AND ABSTRACT INTERPRETATION

6.1 *Introduction*

Modern application development practices pose immense challenges to the realization of the promise of static analysis. Although static analysis techniques can precisely reason about straightforward, imperative programs; modern applications are no longer batch applications. To improve productivity and portability, software engineers increasingly rely on large, complex libraries and frameworks. In particular, frameworks are difficult to analyze due to their extensive use of reflection and metaprogramming, as well as their extreme flexibility driven by external configuration artifacts. Chapter 5 describes the specific challenges posed by these aspects of modern application development. To provide a self-contained motivation for the work presented in this chapter, we will briefly review the challenges posed by frameworks as they are the focus of the work presented here.

Consider the invented code fragment in Figure 6.1, which exemplifies code patterns commonly found in frameworks. Here `start` is part of the application, whereas `delegate`, `init`, and `main` are provided by the framework. The framework first calls `init`, which parses an application-specific XML configuration file, then constructs an `AppContext` to hold the framework state. This state consists of a `delegates` map, which maps names found in the configuration file to class names, and the class name of the application's entry point (field `entryPoint`). The framework reflectively invokes the application's entry point on line 18. When the application calls `delegate` from `start`, the framework uses `delegates` to reflectively instantiate the class associated with the `delegateName` argument and then reflectively invokes the `handle()` method on the newly constructed object. Thus,

```
1  // FRAMEWORK
2  AppContext init(String configFile) {
3    XMLDocument config = XML.parseFile(configFile);
4    AppContext ctxt = ...;
5    for(Node n : config.getNode("delegates"))
6      ctxt.delegates.put(n.get("name"), n.get("class"));
7    // ...
8    ctxt.entryPoint = config.getNodeString("entryPoint");
9    return ctxt;
10 }
11 Object delegate(AppContext ctxt, String delegateName) {
12   String delegateClass = ctxt.delegates.get(delegateName);
13   Object d = Class.forName(delegateClass).newInstance();
14   return d.getClass().getMethod("handle").invoke(d);
15 }
16 void main() {
17   AppContext ctxt = init("conf.xml");
18   Class.forName(ctxt.entryPoint).getMethod("start").invoke(ctxt);
19 }
20 // APPLICATION
21 void start(AppContext ctxt) {
22   while(true) {
23     String request = Network.accept();
24     Object resp;
25     if(request == "ping") {
26        resp = delegate(ctxt, "ping");
27     } else {
28        resp = delegate(ctxt, "pong");
29     }
30     Network.send(resp);
31   }
32 }
```

Figure 6.1: A motivating example demonstrating the difficulty in analyzing framework applications. Specifically, key control-flow decisions in `delegate` and `main` depend on the contents of `conf.xml`.

the callees on lines 14 and 18 are determined entirely by the configuration file, which is opaque to standard call-graph construction algorithms.

To analyze `delegate` and the reflective method invocation on line 18, analysis authors can choose to: 1. unsoundly ignore the reflective call, 2. be extremely imprecise, e.g., by allowing reflective calls to resolve to *any* method, or 3. based on the contents of `conf.xml`, build an application-specific model of the framework behavior. Option 1 misses most of the application's behavior, yielding many false negatives; in Figure 6.1, *none* of the application will be analyzed. Option 2 has the opposite problem: many false positives and infeasible control-flow paths. Finally, option 3 requires significant manual effort. Although this effort can be alleviated with recent techniques (e.g., [174, 23]) creating a model generator for a framework is itself a monumental task.

Other analysis techniques also struggle with framework-based applications. Given a concrete configuration file, the framework methods in Figure 6.1 follow only one execution path, suggesting a partially-concrete approach, such as concolic execution [167, 86, 168]. However, the infinite "accept" loop in `start` is challenging even for state-of-the-art concolic executors. Finitization of the loop can yield false negatives, and some execution engines may fail to terminate. In contrast, a static analysis built using the monotone framework [102] or abstract interpretation [47, 49, 51] can soundly approximate the infinite loop by computing a least fixed point over a series of equations.

A key insight of our approach is that many applications and frameworks follow this pattern: framework implementations are difficult to analyze statically, but large parts are *statically executable* given a concrete, application-specific configuration file. Statically executable refers to a program fragment that: a) can be completely and deterministically evaluated at analysis time, and b) will yield the same program state after evaluation at both runtime and analysis time; the `init` method is an example of statically executable code. Conversely, application code contains unbounded control-flow paths. As a result, a one-size-fits-all approach to program analysis is unwise for framework-based applications.

This chapter presents Concerto, a system for soundly combining *mostly*-concrete interpretation, an extension to concrete interpretation we introduce and formalize in this chapter, and abstract interpretation. By combining these two techniques, Concerto leverages the strengths of both approaches while avoiding their weaknesses. Concerto analyzes framework implementations using mostly-concrete interpretation, and application code using abstract interpretation. Mostly-concrete interpretation supports nondeterminism and over-approximation of sets of values, ensuring our combined interpretation is sound while still terminating.

Our combined interpreter is itself an abstract interpreter that operates over a combined domain of abstract and mostly-concrete states. By formalizing our combined approach within the theory of abstract interpretation (AI), we can directly use techniques from the AI literature to prove soundness, termination, etc. Concerto differs from partial evaluation, as the abstract and concrete interpreters may yield into one another on demand, which allows greater concrete execution within framework code (we illustrate this point further in Section 6.2). Concerto is *analysis agnostic* and can be used with any analysis that satisfies a modest set of conditions. It is *provably sound*: integrating any sound abstract interpretation that satisfies these conditions into Concerto yields a sound, combined analysis. In addition, we have shown that abstract interpretations that satisfy a small, additional set of conditions can *provably* expect equal or greater precision when used with Concerto.

Key to our approach is the observation that framework code does not directly manipulate application state and vice versa. This *state separation* allows Concerto to partition a program state into two disjoint representations: a mostly-concrete representation used to model the framework state, and an abstract representation for application state. The mostly-concrete component of Concerto may therefore manipulate its portion of the program state while remaining agnostic about the abstract representation used by the abstract interpretation component, and vice versa.

We have implemented an initial proof of concept of Concerto for a subset of Java.

We have demonstrated the flexibility of CONCERTO by incorporating three different analyses with different abstract domains into this prototype. We found that using CONCERTO significantly increased precision across all analyses when applied to a difficult-to-analyze framework implementation.

6.2 *Overview*

We will first informally describe how CONCERTO operates on the program in Figure 6.2, which is written in a simple language we will call MAP. Among other features, MAP supports I/O, reflection, and maps which are sufficient to illustrate the core of our technique. This program and language will also serve as our running example as we formalize our approach in the remainder of this chapter. We will later formalize a language parameterized by base constants and operations into which we can embed the MAP language.

In Figure 6.2, the framework code is in the left column and the application code in the right. On lines 2–9, the framework opens an (application-provided) configuration file, and creates a map `m` from application-specific identifiers to procedure names. The application start point, `s`, is then called with `m` as its argument. The only other framework code is `dispatch`, which uses the map produced in `main` to look up a procedure name associated with `k` and invoke the named method. The `invoke` intrinsic reflectively calls the procedure named by the first argument with the remaining arguments. The application-specific logic is implemented in the procedures `s`, `f`, `g`, `h`, and `i`.

Although the program, framework, and language are significantly simpler than the full Java language and the example shown in Figure 6.1, they together pose many of the same challenges. In particular, the reflective operation in `dispatch` depends on information in a configuration file stored in a map. Many difficult-to-analyze framework features, such as dependency injection [77], Android Intents [15], or Servlet dispatch objects (see Chapter 5) follow a similar pattern.

This program exemplifies the *state separation* hypothesis, which requires that the application state is opaque to the framework implementation and similarly for framework

```
1  proc main() {
2    f = open("config");
3    m = empty;
4    k = read f;
5    while(k != "") {
6      v = read f;
7      m = set m k v;
8      k = read f;
9    }
10   // [m ↦ ["b" ↦ "f","a" ↦ "i"]]
11   s(m);
12 }
13 proc dispatch(k, arg, m) {
14   // from g:
15   // [m ↦ ["b" ↦ "f","a" ↦ "i"], arg ↦ {+}]
16   // from h:
17   // [m ↦ ["b" ↦ "f","a" ↦ "i"], arg ↦ {−}]
18   callee = get m k
19   invoke(callee, arg, m);
20 }
```

config
b
f
a
i

```
21 proc s(m) {
22   x = ⋆;
23   while(x >= 0) {
24     g(x, m);
25     x = ⋆;
26   }
27   h(x, m);
28 }
29 proc g(p, m) {
30   p = p + 1;
31   // [m ↦ ["b" ↦ "f","a" ↦ "i"], p ↦ {+}]
32   dispatch("b", p, m);
33 }
34 proc f(p, m) {
35   if(p <= 0) {
36     error();
37   } else {
38     print(p);
39   }
40 }
41 proc h(q, m) {
42   dispatch("a", -4, m);
43 }
44 proc i(q, m) {
45   print(q);
46 }
```

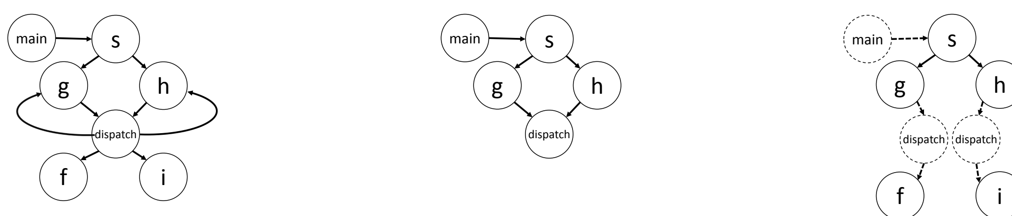
Figure 6.2: Motivating example. The framework implementation (`main` and `dispatch`) uses many of the same implementation idioms found in Figure 6.1. The comments in green show the abstract and mostly-concrete states that reach those program points during combined execution.

state and application code. Thus, the application state (in this case, the integer arguments to `dispatch`) is not interrogated or manipulated by the framework, only threaded through the `dispatch` procedure and then passed back into application procedures. Similarly, the framework state encapsulated in the dispatch map `m` is not directly manipulated by the application, only threaded through application code to calls back into `dispatch`.

In our experience, this hypothesis applies to most real-world implementations; to promote reusability, applications and frameworks rarely directly manipulate each other's state, instead communicating via method calls on opaque interface types. To validate this hypothesis, we performed an informal evaluation of two large Java web frameworks used by programs in the LEGATO evaluation suite, Spring [2] and Struts [76]. The majority of framework state is contained in non-`public` object fields. We found that across the two frameworks only 0.5% (64/12304) of fields had `public` visibility and thus the application cannot mutate the vast majority of framework state. Our experience with other frameworks suggests this pattern is the norm. Similarly, frameworks do not directly mutate application state; to maximize flexibility, frameworks avoid depending on predetermined field names or class layouts, instead relying on well defined interfaces to communicate with the application. Together, these facts suggest that modern Java framework implementations are a natural fit for our state separation hypothesis.

CONCERTO exploits this state separation to thread abstract values produced by the abstract interpreter through concrete interpretation and the abstract interpretation may do the same for concrete values produced by concrete interpretation. (CONCERTO also includes support for the rare cases where this hypothesis does not apply, see Section 6.8.4.) Section 6.3.1 formalizes a type-based state separation that is natural in languages like Java.

Without additional knowledge about the program in Figure 6.2, a standard abstract interpretation not integrated with CONCERTO must make worst-case assumptions about `read`, and thus use an extremely imprecise abstraction of the framework state in `m`. As a result, analysis of `dispatch` would conclude that `invoke` may call any procedure. Thus,



(a) Sound but Imprecise Call-Graph (b) Unsound Call-Graph (c) Call-Graph with Concerto

Figure 6.3: Call graphs produced by different analysis schemes. In (a), calls from `dispatch` to `main` and `s` can be ruled out by matching argument arities. In (c), procedures executed (mostly-)concretely are given a dashed outline.

plain abstract interpretation cannot rule out that a negative argument may flow from `h` through `dispatch` to `f` and that the `error()` statement is reachable. On the other hand, ignoring `invoke` as though it is a no-op ignores important application behavior. The call graphs for these two situations are illustrated in Figures 6.3a and 6.3b, respectively.

Suppose now that we have the following domain knowledge about our program:

1. The contents of the file `config` are available at analysis time and will not change between analysis time and program runtime.

2. `config` has the contents shown in Figure 6.2

This information ensures that `error()` is never executed: `dispatch` will always call `f` with the positive argument passed to it by `g`. However, even if an abstract interpretation has this information, verifying that `error()` is unreachable requires an extremely precise semantics and representation for maps. This precision can be achieved in this simple language, but, in practice, frameworks use much more complicated data structures and abstractions, making precise analysis via pure abstract interpretation unlikely. In contrast, Concerto integrated with a simple signedness analysis can prove that the `error()` statement is unreachable, using a process we briefly sketch below.

6.2.1 *Analyzing the Example*

Concerto begins analysis of the program by concretely executing `main()`. Due to the domain-specific knowledge described above, the initialization loop is statically executable. Concerto opens the file `"config"` and runs the loop to completion. When the loop terminates, `m` holds the map $[$`"b"` $\mapsto$ `"f"`, `"a"` $\mapsto$ `"i"`$]$. We stress that Concerto uses no application- or analysis-specific logic here: Concerto simply performs concrete interpretation, opening the listed file and executing the loop.

At the call to the application entry point `s` on line 11, Concerto switches to abstract interpretation, in this example a signedness analysis. A key assumption of Concerto is that framework code, once given a concrete configuration, is almost entirely statically executable. In contrast, application code may deal with nondeterministic inputs, giving rise to unbounded loops like the one on lines 23–26. Although concrete execution will naturally fail to terminate on the loop in `s`, abstract interpretation can easily over-approximate the loop.

When switching to abstract interpretation, Concerto transforms the concrete program state at the call-site to the abstract representation used by the abstract interpreter. We describe this process in more detail in Section 6.4.3. In this example, the signedness analysis begins in the abstract state: $[m \mapsto [$`"b"` $\mapsto$ `"f"`, `"a"` $\mapsto$ `"i"`$]]$. The abstract interpreter has *not* abstracted away the framework state; instead, the signedness analysis has reused the concrete value directly for the value of m. However, the abstract interpreter does *not* need to implement concrete map semantics or otherwise "understand" this representation. Due to the state separation hypothesis described above, any map operations on `m` occur in framework code, which is *not* analyzed using abstract interpretation.

In the while loop of `s`, the signedness analysis analyzes the call to `g`, which itself calls the framework's `dispatch` procedure in the abstract state $[m \mapsto [$`"b"` $\mapsto$ `"f"`, `"a"` $\mapsto$ `"i"`$], p \mapsto \{+\}]$. For this call, Concerto switches from abstract interpretation to *mostly-*concrete interpretation. Concerto *cannot* soundly switch back to fully-concrete inter-

pretation because the above abstract state cannot be concretized to a single concrete state (or even a finite set of states): p abstracts the infinite set of all positive integers. To avoid materializing an infinite set of concrete states, our mostly-concrete interpretation supports abstractions of infinite sets of values. To represent the infinite set of possible values of `p`, the mostly-concrete interpreter reuses `p`'s abstract value in the abstract interpreter, i.e., $\{+\}$. Concerto is agnostic to the domain of abstract values; had the analysis chosen instead to represent integers with, say, intervals, the mostly-concrete interpreter would also use intervals. As with the embedding of concrete maps into abstract states, the mostly-concrete interpreter does not need semantics for primitive operations over the signedness domain: integer operations on the application state only occur in application code which is *not* executed mostly-concretely.

As the value of `m` in the abstract caller state is $[$`"b"` $\mapsto$ `"f"`, `"a"` $\mapsto$ `"i"`$]$, this value can be directly reused in the callee mostly-concrete state. Hence, mostly-concrete interpretation has no problem concretely evaluating this call to `dispatch` with `"b"` and `m` to determine that control should transfer back to the application by calling `f` with `p`.

At this final call back into the application, Concerto once again switches to abstract interpretation, transforming the mostly-concrete state into an abstract state. The value of `arg` in the mostly-concrete state is $\{+\}$, which becomes the value of `p` in the abstract interpreter. Using this abstract value, the abstract interpreter can prove the true branch of the conditional is never taken. Analysis of the call to `h` on line 27 proceeds similarly, again using mostly-concrete interpretation to precisely resolve the `dispatch` call. The final call-graph used during combined interpretation is shown in Figure 6.3c.

The above process bears many similarities to partial evaluation [82, 140] where the configuration file is treated as a static input. A sufficiently powerful partial evaluator that supports metaprogramming (e.g., [177]), could produce a residual program from Figure 6.2 that a signedness analysis could verify. However, suppose `g` used dynamic input to choose between `"b"` and another procedure name as the argument to `dispatch`. In this scenario, the partial evaluator would *not* fully reduce the body of `dispatch`, making the

signedness analysis imprecise. This scenario also illustrates the benefit of Concerto's on-demand interleaving of mostly-concrete and abstract interpretation. If Concerto performed all concrete execution ahead-of-time as a preprocessing step for abstract interpretation, it would suffer from the same imprecision as partial evaluation. In contrast, provided the abstract interpretation faithfully tracks the two possible procedure names, the mostly-concrete interpreter can precisely resolve the `invoke` operation to the two potential callees.

6.2.2 *Outline*

The remainder of this chapter formalizes and elaborates on the process sketched above. Section 6.3 defines an intraprocedural language we will use and extend throughout the chapter. Our formal language makes explicit the state separation hypothesis. We define a concrete semantics for this language against which we prove Concerto sound, and describe the expected definition of abstract interpreters.

Section 6.4 describes our main contribution: the combination of concrete and abstract interpretation. Section 6.4.1 demonstrates how a naïve combination of concrete and abstract interpretation yields a sound basis for combined interpretation, but one that is impractical to implement due to the difficulty of concretizing abstractions of infinite sets of values, as was the case with $\{+\}$ above. Section 6.4.2 defines *mostly*-concrete interpretation which can handle abstractions of infinite sets of values. Section 6.4.3 shows how combining abstract and *mostly*-concrete interpretation yields a sound interpretation, and formalizes how mostly-concrete states are transformed into abstract states, and vice versa. Finally, Section 6.4.4 defines a set of sufficient conditions for Concerto to match or exceed the precision of an abstract interpretation.

Section 6.5 briefly discusses how we extend our formalism to procedures. Section 6.6 describes a particular iteration strategy that is natural in practice and Section 6.7 sketches how we ensure termination (while retaining soundness) under this iteration strategy

$$
\begin{array}{rcl}
\ell & ::= & \ell_a \mid \ell_f \\
astmt^{\ell} & ::= & \ell_a : astmt \\
astmt & ::= & \texttt{goto}\ \ell \mid x = aexp \mid \texttt{if}\ x \texttt{ <=> } y\ \texttt{goto}\ \ell \\
aexp & ::= & y \mid bc_a \mid aop(v_1, \ldots, v_n)
\end{array}
\qquad
\begin{array}{rcl}
prog & ::= & (fstmt^{\ell} \mid astmt^{\ell})^* \\
fstmt^{\ell} & ::= & \ell_f : fstmt \\
fstmt & ::= & \texttt{goto}\ \ell \mid x = fexp \mid \texttt{if}\ x \texttt{ <=> } y\ \texttt{goto}\ \ell \\
fexpr & ::= & x \mid bc_f \mid fop(v_1, \ldots, v_n)
\end{array}
$$

Figure 6.4: Intraprocedural grammar, parameterized by language-specific choices for bc_f, *fop*, bc_a, and *aop*.

using widening. We then discuss our prototype implementation (Section 6.8), the results of initial case studies (Section 6.9), and then conclude in Section 6.10.

6.3 *Preliminary Definitions*

6.3.1 *Language Definition*

Our core language is a simple imperative language with conditional/unconditional goto, variable assignments, and constants and primitive operations over a set of types. We formalize a *type-based* state separation by partitioning this set of types into application and framework types, and restricting all operations on framework types to framework code and similarly for application types.

The formal grammar is given in Figure 6.4. We assume two disjoint families of types $\mathcal{A}$ (for application) and $\mathcal{F}$ (for framework) and, with a slight abuse of notation, will use $\mathcal{A}$ (respectively $\mathcal{F}$) as a metavariable to range over the types in $\mathcal{A}$ (respectively $\mathcal{F}$). Every variable is given a type drawn from one of these families. bc_f and *fop* range over base constants and primitive operations respectively for types in $\mathcal{F}$, and bc_a and *aop* do the same for $\mathcal{A}$. All operations in *fop* have type $(\mathcal{F} \times \ldots \times \mathcal{F}) \to \mathcal{F}$, and similarly for *aop* and $\mathcal{A}$. The <=> nonterminal ranges over comparison operators.

In our language, ℓ_f label framework statements, and similarly for ℓ_a and application statements. The label of a statement determines what operations may be performed by

that statement: *fop*s and bc_f may only appear in code labeled with ℓ_f, and similarly for *aop*, bc_a and ℓ_a. Further, we require that comparisons in statements labeled with ℓ_a can only compare values of types in $\mathcal{A}$, and similarly for ℓ_f and $\mathcal{F}$. However, statements labeled ℓ_f may move values between variables with type $\mathcal{A}$, and vice versa for statements labeled ℓ_a. In other words, framework code must treat values of type $\mathcal{A}$ ("application values") opaquely, threading them through to statements labeled ℓ_f. Similarly, statements labeled ℓ_a must treat $\mathcal{F}$ values opaquely. This syntactic restriction models the state separation hypothesis described in Section 6.2. (In practice, this strict separation may be violated by, e.g., primitive types like `int`, library types, etc.; our implementation has special support for these shared types as described in Section 6.8.4.)

The full operational semantics (found in Figure 6.5) are defined in terms of the following denotations and value domains, which we will use throughout the remainder of the paper. Let V_a be the set of all values with a type in $\mathcal{A}$ and similarly for V_f and $\mathcal{F}$. Every *aop* has a denotation $[\![aop]\!] : (V_a \times \dots V_a) \to V_a$; we assume that *aop*s are deterministic. In contrast, some *fop* operations may produce values that depend on the value of some environment model $\mathcal{E}$, which models nondeterminism due to file contents, network requests, etc. Formally, each *fop* has a denotation $[\![fop]\!] : \mathcal{E} \times V_f \times \dots \times V_f \to \mathbb{P}(V_f \times \mathcal{E})$, where $\langle v', E' \rangle \in [\![fop]\!](E, v_1, \dots, v_n)$ means executing *fop* in environment $E \in \mathcal{E}$ with arguments $v_1, \dots, v_n$ produces a new environment model E' and result v'. To simplify presentation, we require that denotations are total functions; in practice, we assume that the denotations gracefully handle runtime type errors (e.g., by returning a sentinel error value, halting execution, etc.). Finally, we assume that $[\![<=>]\!]$ denotes into a binary relation over values of the appropriate type, $[\![<\neq>]\!]$ is the negation of $[\![<=>]\!]$, and $[\![bc_f]\!]$ and $[\![bc_a]\!]$ produce values in V_f and V_a respectively corresponding to the interpretation of those constants.

Although the result of *fop*s depend on the current environment E, in some cases we may have *a priori* information such that a seemingly nondeterministic *fop*, i.e., reading a file, is *effectively deterministic*. For example, in Section 6.2, we exploited domain specific information about runtime contents of the configuration file to precisely execute the

$$\langle e, s, \ell \rangle \to \langle e', s', \ell' \rangle$$

$$\text{GOTO}\ \frac{prog[\ell] = \texttt{goto}\ \ell'}{\langle e, s, \ell \rangle \to \langle e, s, \ell' \rangle} \qquad \text{ASSIGN}\ \frac{prog[\ell] = x \texttt{ = } y \qquad succ(\ell, \ell')}{\langle e, s, \ell \rangle \to \langle e, s[x \mapsto s[y]], \ell' \rangle}$$

$$\text{CONST}\ \frac{prog[\ell] = x \texttt{ = } c \qquad c \in \mathrm{bc}_a \cup \mathrm{bc}_f \qquad succ(\ell, \ell')}{\langle e, s, \ell \rangle \to \langle e, s[x \mapsto [\![c]\!]], \ell' \rangle}$$

$$\text{IF-TRUE}\ \frac{prog[\ell] = \texttt{if}\ x \texttt{ <=> } y\ \texttt{goto}\ \ell' \qquad s[x][\![\texttt{<=>}]\!]s[y]}{\langle e, s, \ell \rangle \to \langle e, s, \ell' \rangle}$$

$$\text{IF-FALSE}\ \frac{prog[\ell] = \texttt{if}\ x \texttt{ <=> } y\ \texttt{goto}\ \ell'' \qquad \neg(s[x][\![\texttt{<=>}]\!]s[y]) \qquad succ(\ell, \ell')}{\langle e, s, \ell \rangle \to \langle e, s, \ell' \rangle}$$

$$\text{AOP}\ \frac{prog[\ell] = x \texttt{ = } aop\ (y_1, \ldots, y_n) \qquad [\![aop]\!](s[y_1], \ldots, s[y_n]) = v \qquad succ(\ell, \ell')}{\langle e, s, \ell \rangle \to \langle e, s[x \mapsto v], \ell' \rangle}$$

$$\text{FOP}\ \frac{prog[\ell] = x \texttt{ = } fop\ (y_1, \ldots, y_n) \qquad \langle v, e' \rangle \in [\![fop]\!](s[y_1], \ldots, s[y_n]) \qquad succ(\ell, \ell')}{\langle e, s, \ell \rangle \to \langle e', s[x \mapsto v], \ell' \rangle}$$

Figure 6.5: Operational semantics for the intraprocedural language.

initialization loop. We account for this knowledge by allowing hypotheses on the domain of $\mathcal{E}$. For example, if $\mathcal{E}$ models file-system contents, and we have *a priori* knowledge that a file $\mathtt{f}$ always has content $\mathtt{c}$ at runtime, we can restrict $\mathcal{E}$ to include only models where the file $\mathtt{f}$ has contents $\mathtt{c}$.

Throughout the rest of this chapter, we assume that we are operating on some arbitrary program written in this language and that the relations *pred* and *succ* are defined with the obvious definitions and there is a map *prog* from labels to unlabeled statements.

Example 6.3.1 (MAP Language)**.** We can encode the MAP language and state separation of Section 6.2 in this language framework as follows. Control-flow constructs (`if`/`while`) can be encoded using the `goto` representation defined in Figure 6.4; we defer discussion of procedures to Section 6.5.

We take $\mathcal{A} = \{\mathtt{int}\}$, i.e., the type of machine integers, with $\mathtt{bc}_a$ defined to be integer constants, $\mathtt{aop}$ to be the usual arithmetic operations, and V_a as machine integers. We next define $\mathcal{F} = \{File, Str, \mathcal{M}\}$, where $\mathcal{M}$ is the domain of maps from strings to strings, and instantiate *fop* with the following:

$$\mathtt{open} : \mathsf{Str} \to \mathsf{File} \qquad \mathtt{read} : \mathsf{File} \to \mathsf{Str} \qquad \mathtt{set} : \mathcal{M} \times \mathsf{Str} \times \mathsf{Str} \to \mathsf{Str} \qquad \mathtt{get} : \mathcal{M} \times \mathsf{Str} \to \mathsf{Str}$$

We define $\mathtt{bc}_f$ as the set of literal string constants and $\mathtt{empty} : \mathcal{M}$ which is an empty map. Finally, we take $V_f = \mathsf{STR} \cup \mathit{FileContents} \cup (\mathsf{STR} \to \mathsf{STR})$, where STR is the set of string values and *FileContents* is a finite stream of STR values. Given the syntactic constraints on where *aop*s and *fop*s may appear, this instantiation encodes that the application may not manipulate the framework dispatch map `m`, nor may the framework manipulate integers received from the application.

Finally, the domain of environment models $\mathcal{E}$ is a map from strings to file contents, i.e., $\mathsf{STR} \to \mathit{FileContents}$. We encode the information about the configuration file by requiring that: $\mathcal{E} = \{e \mid e \in \mathsf{STR} \to \mathit{FileContents} \land e[\texttt{"config"}] = \langle \texttt{"b"}, \texttt{"f"}, \texttt{"a"}, \texttt{"i"} \rangle\}$ □

6.3.2 *Concrete Properties*

Concerto targets abstract interpretations where the approximated concrete property is the set of reaching concrete states that may occur during program execution. In the intraprocedural language described thus far, a concrete state is a valuation for the variables in the program plus an environment model. Formally, a concrete state is defined as: $S = \mathcal{E} \times (X \to V)$ where $V = V_a \cup V_f \cup \{\bot_V\}$, X is the domain of variables appearing in a program, $\mathcal{E}$ is type of environment models described in Section 6.3.1, and $\bot_V$ is a sentinel "uninitialized" value. We assume the type system is sound, i.e., for any concrete state s arising during execution: $\mathtt{type}(x) \in \mathcal{F} \Rightarrow s[x] \in V_f$ and $\mathtt{type}(x) \in \mathcal{A} \Rightarrow s[x] \in V_a$, and that all variables must be defined before use, i.e., a program never observes $\bot_V$.

Next, define a domain of *flow edges* $\mathcal{L}$ as: $\mathcal{L} = \{\ell^\circ \rightsquigarrow \ell^\bullet\} \cup \{p^\bullet \rightsquigarrow \ell^\circ \mid p \in \mathtt{pred}(\ell)\}$, where ℓ° and $\ell^\bullet$ respectively denote the "entrance to" and "exit from" ℓ. Elements $\ell^\circ \rightsquigarrow \ell^\bullet$ of $\mathcal{L}$ correspond to the flow of program control through the statement labeled ℓ, whereas an element $p^\bullet \rightsquigarrow \ell^\circ$ corresponds to the flow from some predecessor p to ℓ. When convenient, we will abbreviate $\ell^\circ \rightsquigarrow \ell^\bullet$ as simply ℓ. We will use $\vec{\ell}$ to represent an arbitrary element of $\mathcal{L}$.

Our domain of concrete properties is $R = \mathcal{L} \to \mathbb{P}(S)$, which forms a complete lattice, defined pointwise over edges; the powerset of states also forms a complete lattice with the usual definitions.

We now define the concrete semantic function, $F : R \to R$, the least fixed-point of which is a complete set of reaching states for a program. That is, for every $\vec{\ell} \in \mathcal{L}$, if $\langle s, E' \rangle \in (\mathit{lfp}\, F)[\vec{\ell}]$ then there is some execution that flows through edge $\vec{\ell}$ yielding $\langle s, E' \rangle$. The full definition of F is given in Figure 6.6. As is standard, the definition of F closely mirrors the operational semantics of the language shown in Figure 6.5. We assume program execution begins at a single, distinguished label s. Program start is modeled with the second term in the definition of $F(r)[\ell^\circ \rightsquigarrow \ell^\bullet]$: $\iota_{\mathcal{E}}$ is a set of possible initial environments, and ι_S is a distinguished start state which maps all variables to $\bot_V$.

$$F(r)[\ell^\circ \rightsquigarrow \ell^\bullet] = \bigsqcup_{\substack{p\in pred(\ell)\\ in\in r[p^\bullet\rightsquigarrow\ell^\circ]}} step^F(in,\ell) \sqcup \begin{cases} \bigsqcup_{e\in\iota_{\mathcal{E}}} step^F(\langle\iota_S, e\rangle,\ell) & \ell = s \\ \emptyset & o.w. \end{cases}$$

$$F(r)[p^\bullet \rightsquigarrow \ell^\circ] = \begin{cases} \{\langle s,E\rangle | \langle s,E\rangle \in r[p^\circ \rightsquigarrow p^\bullet] \wedge s[x][\![\texttt{<=>}]\!]s[y]\} & prog[p] = \texttt{if x <=> y goto } \ell \\ \{\langle s,E\rangle | \langle s,E\rangle \in r[p^\circ \rightsquigarrow p^\bullet] \wedge (s[x][\![\texttt{<}\neq\texttt{>}]\!]s[y])\} & prog[p] = \texttt{if x <=> y goto } \ell' \\ r[p^\circ \rightsquigarrow p^\bullet] & o.w. \end{cases}$$

$$step^F(\langle in, E\rangle,\ell) = \begin{cases} \{\langle in, E\rangle\} & prog[\ell] = \texttt{if} \ldots \vee \texttt{goto} \ldots \\ \{\langle in[x\mapsto in[y]], E\rangle\} & prog[\ell] = x = y \\ \{\langle in[x\mapsto [\![c]\!]], E\rangle\} & prog[\ell] = x = c \\ \{\langle in[x\mapsto [\![aop]\!](in[v_1],\ldots,in[v_n])], E\rangle\} & prog[\ell] = x = aop(v_1,\ldots,v_n) \\ \{\langle in[x\mapsto r], E'\rangle \mid \langle r, E'\rangle \in [\![fop]\!](E, in[v_1],\ldots,in[v_n])\} & prog[\ell] = x = fop(v_1,\ldots,v_n) \end{cases}$$

Figure 6.6: Concrete semantic function. c is any constant of type $\mathcal{A}$ or $\mathcal{F}$ and $[\![c]\!]$ is its corresponding denotation.

6.3.3 *Abstract Properties*

Concerto is designed to combine mostly-concrete execution with an abstract interpretation defined as follows. We assume that sets of reaching concrete states are over-approximated by the complete lattice $\widehat{S}$ with ordering and least upper bound operator $\sqsubseteq_{\widehat{S}}$ and $\sqcup_{\widehat{S}}$ respectively. (We will use $\sqsubseteq_D$ to indicate a partial order on a domain D.) The domain of abstract properties, $\widehat{R} = \mathcal{L} \to \widehat{S}$, also forms a complete lattice defined by the pointwise extension of $\sqsubseteq_{\widehat{S}}$ and $\sqcup_{\widehat{S}}$. We further assume that the domain $\widehat{R}$ forms a Galois connection with the domain of concrete properties R, defined by the abstraction and concretization functions α_A and γ_A. We use the standard notation $R \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} \widehat{R}$ to denote this connection. The abstract semantics of the abstract interpretation are given by a monotone abstract semantic function $\widehat{F} : \widehat{R} \to \widehat{R}$. We assume that this function is a sound abstraction of F according to the above Galois connection, i.e., $\alpha_A \circ F \sqsubseteq_{R\to\widehat{R}} \widehat{F} \circ \alpha_A$,[1]

[1]Equivalently, $\alpha_A \circ F \circ \gamma_A \sqsubseteq_{\widehat{R}\to\widehat{R}} \widehat{F}$ or $F \circ \gamma_A \sqsubseteq_{\widehat{R}\to R} \gamma_A \circ \widehat{F}$.

$$\widehat{F}(\widehat{r})[\ell^\circ \rightsquigarrow \ell^\bullet] = \mathtt{step}^+(\bigsqcup_{p \in \mathtt{pred}(\ell)} \widehat{r}[p], \ell) \sqcup \begin{cases} \mathtt{step}^+(\bot, \ell) & \ell = s \\ \bot & o.w. \end{cases}$$

$$\widehat{F}(\widehat{r})[p^\bullet \rightsquigarrow \ell^\circ] = \begin{cases} \widehat{r}[p^\circ \rightsquigarrow p^\bullet][x \mapsto \mathsf{lop} \cap \{0,+\}] & \text{if } \mathit{prog}[s] = \texttt{if } x \texttt{ >= } 0 \texttt{ goto } \ell \wedge \mathsf{lop} \cap \{0,+\} \neq \emptyset \\ \bot & \text{if } \mathit{prog}[s] = \texttt{if } x \texttt{ <= } 0 \texttt{ goto } \ell' \wedge \mathsf{lop} \cap \{-,0\} = \emptyset \\ \ldots & \\ \widehat{r}[p] & o.w. \end{cases}$$

$$\mathtt{step}^+(\widehat{\mathtt{in}}, \ell) = \begin{cases} \widehat{\mathtt{in}}[x \mapsto \widehat{\mathtt{in}}[y]] & \mathit{prog}[\ell] = x \texttt{ = } y \\ \widehat{\mathtt{in}}[x \mapsto \widehat{\mathtt{in}}[y] \widehat{[\![\mathit{aop}]\!]} \widehat{\mathtt{in}}[z]] & \mathit{prog}[\ell] = x \texttt{ = } y \ \mathit{aop} \ z \\ \widehat{\mathtt{in}}[x \mapsto \{\mathtt{sign}(\mathsf{N})\}] & \mathit{prog}[\ell] = x \texttt{ = } \mathsf{N} \\ \widehat{\mathtt{in}} & \mathit{prog}[\ell] = \texttt{goto} \ldots \vee \mathit{prog}[\ell] = \texttt{if} \ldots \\ \widehat{\mathtt{in}}[x \mapsto \top_F] & \mathit{prog}[\ell] = x \texttt{ = } \mathsf{bc}_f \vee \mathit{prog}[\ell] = x \texttt{ = } \mathit{fop}(v_1, \ldots, v_n) \end{cases}$$

Figure 6.7: The abstract semantics for our running signedness example. In the above, we use an infix notation for *aop* as we assume all integer operations are binary. In the $\mathtt{step}^+$ function, N is an integer constant; in the definition of $\widehat{F}$, lop is $\widehat{r}[p^\circ \rightsquigarrow p^\bullet][x]$ and s is the distinguished start label.

whence by [49] we have that $\alpha_A(\mathit{lfp}\, F) \sqsubseteq_{\widehat{R}} \mathit{lfp}\, \widehat{F}$. In other words, the reaching states computed by the least fixed point of F are soundly over-approximated according to α_A by the least fixed point of $\widehat{F}$.

Thus, an abstract interpretation is defined by the 7-tuple $\langle \widehat{R}; \widehat{S}; \widehat{F}; \alpha_A; \gamma_A; \sqcup_{\widehat{S}}; \sqsubseteq_{\widehat{S}} \rangle$ where:

$$\widehat{R} = \mathcal{L} \to \widehat{S} \qquad R \underset{\alpha_A}{\overset{\gamma_A}{\leftrightarrows}} \widehat{R} \qquad \alpha_A \circ F \sqsubseteq_{R \to \widehat{R}} \widehat{F} \circ \alpha_A$$

Example 6.3.2 (Signedness Analysis)**.** The abstract state for the intraprocedural signedness analysis discussed in Section 6.2 is $\widehat{S} = X_f \to \mathbb{P}(V_f)^\top \times X_a \to \mathbb{P}(\{-, 0, +\})$, and α_A is

defined as:

$$\alpha_A(r)[\vec{\ell}] = \left\langle \lambda x : X_f.\{s[x] \mid \langle s, e\rangle \in r[\vec{\ell}] \wedge s[x] \neq \bot_V\},\ \lambda x : X_a.\{\mathit{sign}(s[x]) \mid \langle s, e\rangle \in r[\vec{\ell}]\}\right\rangle$$

$$\mathit{sign}(n) = \begin{cases} + & n > 0 \\ 0 & n = 0 \\ - & n < 0 \end{cases}$$

In the above definitions, X_f is the set of program variables with type in $\mathcal{F}$, X_a are those with types in $\mathcal{A}$. Intuitively, α_A abstracts variables of framework type (i.e., files, strings, maps, etc.) with sets of concrete values and abstracts integers with the standard signedness domain. We omit the definition of γ_A as it can be derived from the definition of α_A [49]. The domain $\mathbb{P}(V_f)^\top$ is the powerset domain of concrete values, extended with a special $\top_F$ value, which represents any possible value of type $\mathcal{F}$. We omit the definitions of $\sqsubseteq_{\widehat{S}}$ and $\sqcup_{\widehat{S}}$ as they are standard.

The abstract semantic function $\widehat{F}$ is defined in Figure 6.7. We have included only the comparison rules necessary to verify the program in Figure 6.2. We omit the definitions for $\widehat{[\![+]\!]}$, $\widehat{[\![-]\!]}$, etc. However $\widehat{[\![+]\!]}$ is defined such that $\{0, +\}\widehat{[\![+]\!]}\{+\} = \{+\}$, which again is sufficient to verify the example in Figure 6.2. □

6.4 *Combined Interpretation*

Given the language, semantic functions, and Galois connection defined in Section 6.3, we can define an initial, naïve attempt at combined interpretation. Intuitively, this strawman combination, which we call Concerto$_0$, analyzes framework code by applying the concrete semantic function F at framework statements and the abstract semantic function $\widehat{F}$ at application statements. Concerto$_0$ translates between abstract and concrete states using explicit abstraction and concretization functions. This approach is sound (as proved in the following Section 6.4.1) but ultimately infeasible to implement as it requires materializing infinite sets of states and values. To overcome this limitation, we

extend concrete interpretation to *mostly-concrete* interpretation. Mostly-concrete interpretation avoids materializing infinite sets by using *finite* abstractions of sets of possible values. We then define the combination of mostly-concrete and abstract interpretation used by Concerto and prove it sound. This combination must translate between different state representations. Unlike Concerto_0, we do not use explicit abstraction or concretization functions. Instead we formalize *domain transformers* which soundly translate between state domains but are weaker than a Galois connection. We close by proving when Concerto matches or exceeds the precision of plain abstract interpretation.

6.4.1 Naïve Combination

To motivate the need for mostly-concrete interpretation, we elaborate on the strawman Concerto_0 and enumerate why it is impractical as a basis for combined analysis.

Concerto_0 executes framework code (statements with label ℓ_f) concretely and abstractly interprets application code (statements with label ℓ_a). Our initial attempt at combined interpretation therefore operates over a combined semantic domain that represents reaching states in the framework with sets of concrete states, and reaching states in the application with the abstract state domain $\widehat{S}$. However, the semantic function F, which models concrete execution, operates over the fully-concrete domain R. The combined domain is injected into the fully-concrete domain by applying a concretization function to the abstract states of the combined domain. Symmetrically, to abstractly execute application code with the abstract semantic function $\widehat{F}$, the combined domain is injected into the abstract domain by applying an abstraction function to the reaching concrete state component. After injection and applying both semantic functions, Concerto_0 combines the concrete results at framework statements and abstract results at application statements.

As the combined interpretation uses the highly precise concrete semantics for framework statements (and therefore *fop*s), Concerto_0 can, at least in principle, precisely

analyze framework code. The concrete interpreter may also use any hypotheses on run-time environments to gain further precision. For example, if the framework parses a configuration file as in the example of Section 6.2, the concrete interpreter may simply open and parse the configuration file directly. However, not all *fop*s will be analysis-time deterministic, leading to an explosion in reaching concrete states. Further, Concerto$_0$ relies on an explicit concretization function which cannot be implemented in practice.

We formalize the informal description above as follows. First, we partition the space of $\mathcal{L}$ into two sets, $\mathcal{L}_A = \{\ell_a^\circ \rightsquigarrow \ell_a^\bullet\} \cup \{\ell_a^\bullet \rightsquigarrow \ell^\circ \mid \ell_a \in \mathtt{pred}(\ell)\}$ and $\mathcal{L}_F = \{\ell_f^\circ \rightsquigarrow \ell_f^\bullet\} \cup \{\ell_f^\bullet \rightsquigarrow \ell^\circ \mid \ell_f \in \mathtt{pred}(\ell)\}$. $\mathcal{L}_A$ are flow edges originating in ℓ_a-labeled statements, and symmetrically for $\mathcal{L}_F$ and ℓ_f. We define this combined domain $\overline{R}_0$ and an abstraction function $\alpha_0 : R \to \overline{R}_0$ as:[2]

$$\overline{R}_0 = \mathcal{L}_F \to \mathbb{P}(S) \times \mathcal{L}_A \to \widehat{S} \qquad\qquad \alpha_0(r) = \langle \lambda\vec{\ell} : \mathcal{L}_F.r[\vec{\ell}],\ \lambda\vec{\ell} : \mathcal{L}_A.\alpha_A(r)[\vec{\ell}]\rangle$$

$\overline{R}_0$ is a complete lattice with the standard component-wise definitions of least upper bound and ordering. As α_0 is a monotone, complete join morphism there is some γ_0 such that R and $\overline{R}_0$ form a Galois connection. We further assume that the abstract and concrete states form a Galois connection $\mathbb{P}(S) \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \widehat{S}$ and that $\alpha_A = \dot{\alpha_S}$ and $\gamma_A = \dot{\gamma_S}$, where $\dot{f}$ denotes the pointwise extension of f. Using α_S and γ_S, the injection functions described above are defined as:

$$inj_R(\langle m, \widehat{m}\rangle)[\vec{\ell}] = \begin{cases} m[\vec{\ell}] & \vec{\ell} \in \mathcal{L}_F \\ \gamma_S(\widehat{m}[\vec{\ell}]) & o.w. \end{cases} \qquad inj_{\widehat{R}}(\langle m, \widehat{m}\rangle)[\vec{\ell}] = \begin{cases} \alpha_S(m[\vec{\ell}]) & \vec{\ell} \in \mathcal{L}_F \\ \widehat{m}[\vec{\ell}] & o.w. \end{cases}$$

We can now define Concerto$_0$'s combined interpretation function $C_0 : \overline{R}_0 \to \overline{R}_0$ as:

$$C_0(X) = \left\langle F(inj_R(X))|_{\mathcal{L}_F},\ \widehat{F}(inj_{\widehat{R}}(X))|_{\mathcal{L}_A} \right\rangle \tag{6.1}$$

Theorem 3. C_0 *is sound, i.e.,* $\alpha_0 \circ F \sqsubseteq_{R \to \overline{R}_0} C_0 \circ \alpha_A$.

[2] The original conference version of this work [185] mistakenly omitted the powerset notation on the concrete state component of $\overline{R}_0$ and in the Galois connection below.

Proof. By the assumed soundness of $\widehat{F}$, that $inj_{\widehat{R}} \circ \alpha_0 = \alpha_A$ and that $inj_R \circ \alpha_0$ is extensive. ☐

Theorem 3 establishes C_0 is sound, but that doesn't mean C_0 is a good idea. In fact, it is not. Using C_0 as the basis for combined interpretation is impractical for the following reasons:

(1) *Infinite Sets* In many cases, concretizing an abstract state will yield an infinite number of concrete states. For example, concretizing the abstract state $[m \mapsto [\texttt{"b"} \mapsto \texttt{"f"}, \texttt{"a"} \mapsto \texttt{"i"}], p \mapsto \{+\}]$ at the call to `dispatch` in Section 6.2 yields the following set of concrete states:

$$\bigcup_{e \in \mathcal{E}} \{\langle e, [m \mapsto [\texttt{"b"} \mapsto \texttt{"f"}, \texttt{"a"} \mapsto \texttt{"i"}], p \mapsto n] \rangle \mid n > 0\}$$

The concretization operation cannot be implemented in a meaningful way, as materializing this infinite set is clearly impossible.

(2) *Nondeterminism* Not all framework operations will be deterministic, even with *a priori* knowledge about the program's runtime environment. For example, the exact user input entered is impossible to know at analysis time. Instead, combined interpretation based on C_0 would have to enumerate over all possible values produced in all environments which is impractical from an implementation perspective.

(3) *Exponential Explosion in States* Nondeterministic *fop*s whose denotations return multiple possible results will cause exponential explosions in the number of reaching states. For example, if n states reach an *fop* that produces m unique results, the concrete semantics will generate $n \cdot m$ result states. This problem is similar to the exponential state explosion common in symbolic execution.

Instead of combining abstract and concrete interpretation, Concerto combines abstract and *mostly*-concrete interpretation, which addresses the above limitations. Section 6.4.2 describes the semantics of mostly-concrete interpretation, Section 6.4.3 describes how to combine abstract and mostly-concrete interpretation soundly, and finally Section 6.4.4 gives our precision result for combined interpretation.

6.4.2 Mostly-Concrete Interpretation

Mostly-concrete interpretation introduces the following extensions to concrete interpretation:

Extension 1: The mostly-concrete interpreter supports runtime values that are finite abstractions of (potentially infinite) sets of values. For example, the abstract value $\{+\}$ from Section 6.2 is a finite abstraction of an infinite set of numbers. Thus, when converting from an abstract to mostly-concrete state, Concerto does *not* need to materialize an infinite (or unmanageably large) set of concrete values; the abstract interpretation need only provide these abstractions.

Extension 2: When an *fop* would yield an infinite or unmanageably large set of values, the mostly-concrete interpreter may instead use a special "unknown" value that represents "any possible value." The mostly-concrete semantics extend the fully concrete semantics to soundly handle this value.

Extension 3: The mostly-concrete semantic domain is a map from flow edges to a *single* mostly-concrete state, which itself maps program variables to the abstractions of multiple possible values mentioned in Extension 1 above. Thus, our mostly-concrete domain cannot track relationships between program variables, i.e., it is *non*-relational.

Except for these extensions, the mostly-concrete semantics mirror the concrete semantics (hence the name). Provided all *fop*s are deterministic and environment agnostic,[3] given a deterministic input state, the results of mostly-concrete and concrete interpretation on framework code will coincide.

To represent multiple possible values for variables of type $\mathcal{F}$, the mostly-concrete domain uses powersets of values in V_f extended with a special "unknown" value. For $\mathcal{A}$,

[3]We say an *fop* is environment agnostic if it does not depend on the environment model, i.e., $\forall e_1, e_2 \in \mathcal{E}, v_1, \ldots, v_n \in V_f. \{v \mid \langle v, _\rangle \in [\![fop]\!](e_1, v_1, \ldots, v_n)\} = \{v \mid \langle v, _\rangle \in [\![fop]\!](e_2, v_1, \ldots, v_n)\}$

the abstract interpretation provides an abstraction domain $\widehat{A}$ that satisfies the properties described in Section 6.4.3. Informally, the $\widehat{A}$ domain must be non-relational and path-insensitive. However, the abstract value domain used internally by the AI has no such restriction.

Formally, the domain of reaching mostly-concrete states $\widetilde{R}$ is:

$$\widetilde{R} = \mathcal{L} \to \widetilde{S} \qquad \widetilde{S} = (X_f \to \mathbb{P}(V_f)^\top) \times (X_a \to \widehat{A})$$

As in Example 6.3.2, X_f and X_a are the sets of program variables with types in $\mathcal{F}$ and $\mathcal{A}$ respectively. $\mathbb{P}(V_f)^\top$ is the powerset domain of concrete values in $\mathcal{F}$, extended with $\top_F$, which is the "unknown" value described in Extension 2. The abstractions for variables with types in $\mathcal{A}$ in the mostly-concrete interpreter are drawn from the domain $\widehat{A}$ provided by the AI; $\widehat{A}$ is assumed to form a complete lattice. $\widehat{A}$ may or may not be used internally by the abstract interpreter. The domain of $\widetilde{S}$ and $\widetilde{R}$ form a complete lattice equipped with the standard pointwise join and ordering operators, as well as top and bottom values. We further assume that the analysis defines a monotone complete join-morphism $\alpha_v : \mathbb{P}(V_a) \to \widehat{A}$ that abstracts a set of concrete values of type $\mathcal{A}$ to a value in $\widehat{A}$.

Example 6.4.1 (Signedness Analysis)**.** For the signedness analysis, $\widehat{A}$ is the same domain as used in the abstract interpretation, $\mathbb{P}(\{-, 0, +\})$. α_v is defined as: $\alpha_v(I) = \{\mathit{sign}(i) \mid i \in I\}$ where *sign* is defined as in Example 6.3.2. The join and ordering operators are set union and inclusion respectively. In the example from Section 6.2, Concerto used the abstract value $\{+\}$ for the value of `arg` in `dispatch` which abstracted the set $\{n | n > 0\}$. □

Example 6.4.2 (Pentagons)**.** Suppose instead the abstract interpretation of Section 6.2 had used a relational domain like Pentagons [130], which is the interval domain complemented with symbolic upper bounds. Then, $\widehat{A} = \mathit{Intv}$, where *Intv* is the plain interval

domain [47]. α_v is defined as:

$$lb\, N = \begin{cases} -\infty & \nexists n_0 \in \mathbb{Z}. \forall n \in N. n_0 \leqslant n \\ min(N) & o.w. \end{cases}$$

$$ub\, N = \begin{cases} \infty & \nexists n_0 \in \mathbb{Z}. \forall n \in N. n_0 \geqslant n \\ max(N) & o.w. \end{cases}$$

$$\alpha_v(I) = \begin{cases} \bot & I = \emptyset \\ [lb\, I, ub\, I] & o.w. \end{cases}$$

The join and ordering operators are the standard interval union and inclusion operators. Unlike the above example, we *cannot* reuse the pentagon domain for $\widehat{A}$ for reasons discussed in Section 6.4.3. □

We now define an abstraction function α_F as follows:

$$\alpha_F(r)[\ell] = \left\langle \lambda x : X_f . \mathcal{V}(r[\ell], x),\ \lambda x : X_a . \alpha_v(\mathcal{V}(r[\ell], x)) \right\rangle$$

Where $\mathcal{V}$ is the reaching value set for a set of states and variable, defined as: $\mathcal{V}(S, x) = \{s[x] \mid \langle s, _ \rangle \in S \wedge s[x] \neq \bot_V\}$. α_F approximates a variable x of type $\mathcal{F}$ with exactly the set of (initialized) values of x. Variables of type $\mathcal{A}$ are approximated by applying α_v to the set of reaching values. The approximation for a variable may not depend on the value of other variables, nor the location at which the variable is being approximated.

As α_v is a complete join morphism, α_F is also a complete join-morphism, and thus by [49], there exists some γ_F such that $R \underset{\alpha_F}{\overset{\gamma_F}{\leftrightarrows}} \widetilde{R}$.

In Figure 6.8, we define the mostly-concrete semantic function $I_\top : \widetilde{R} \to \widetilde{R}$. The structure of $I_\top$ closely mirrors that of the concrete semantic function F. Operations and comparisons on values from types in $\mathcal{F}$ use the same operations as in F, but lifted to powersets of values. If one of the operands to a comparison is unknown (i.e., $\top_F$), then

$$I_\top(\widetilde{r})[\ell^\circ \rightsquigarrow \ell^\bullet] = \mathit{step}^\top(\bigsqcup_{p \in \mathit{pred}(\ell)} \widetilde{r}[p^\bullet \rightsquigarrow \ell^\circ], \ell) \sqcup \begin{cases} \mathit{step}^\top(\bot_{\widetilde{S}}, \ell) & \ell = s \\ \bot_{\widetilde{S}} & \text{o.w.} \end{cases}$$

$$I_\top(\widetilde{r})[p^\bullet \rightsquigarrow \ell^\circ] = \begin{cases} \widetilde{r}[p] & \widetilde{\mathcal{FT}}(\widetilde{r}, p^\bullet \rightsquigarrow \ell^\circ) \\ \bot & \text{o.w.} \end{cases}$$

$$\widetilde{\mathcal{FT}}(\widetilde{r}, p^\bullet \rightsquigarrow \ell^\circ) = \begin{cases} \widetilde{s}[x]\widetilde{[\![\texttt{<=>}]\!]}\widetilde{s}[y] & p \in \ell_f \wedge \mathrm{prog}[p] = \texttt{if x<=>y goto } \ell \quad (6.2) \\ \widetilde{s}[x]\widetilde{[\![\texttt{<}\neq\texttt{>}]\!]}\widetilde{s}[y] & p \in \ell_f \wedge \mathrm{prog}[p] = \texttt{if x<=>y goto } \ell' \quad (6.3) \\ \mathrm{true} & \text{o.w.} \quad (6.4) \end{cases}$$

Where $\widetilde{s} = \widetilde{r}[p^\circ \rightsquigarrow p^\bullet]$ and $\widetilde{[\![R]\!]}$ is the lifting of R defined by:

$$\widetilde{v}\widetilde{[\![R]\!]}\top_F \qquad \top_F\widetilde{[\![R]\!]}\widetilde{v} \qquad v \in \widetilde{v} \wedge v' \in \widetilde{v'} \wedge v[\![R]\!]v' \Rightarrow \widetilde{v}\widetilde{[\![R]\!]}\widetilde{v}'$$

$$\mathit{step}^\top(\widetilde{in}, \ell) = \begin{cases} \widetilde{in} & \mathrm{prog}[\ell] = \texttt{if} \ldots \vee \texttt{goto} \ldots \quad (6.5) \\ \widetilde{in}[x \mapsto \widetilde{in}[y]] & \mathrm{prog}[\ell] = x \texttt{ = } y \quad (6.6) \\ \widetilde{in}[x \mapsto \{[\![bc_f]\!]\}] & \mathrm{prog}[\ell] = x \texttt{ = } bc_f \quad (6.7) \\ \widetilde{in}[x \mapsto \widetilde{[\![fop]\!]}(\widetilde{in}[v_1], \ldots, \widetilde{in}[v_n])] & \mathrm{prog}[\ell] = x \texttt{ = } fop(v_1, \ldots, v_n) \quad (6.8) \\ \widetilde{in}[x \mapsto \top_{\widehat{A}}] & \mathrm{prog}[\ell] = x \texttt{ = } aexp \wedge \mathrm{type}(x) = A \quad (6.9) \end{cases}$$

$$\widetilde{[\![fop]\!]}(\widetilde{f_1}, \ldots, \widetilde{f_n}) = \begin{cases} \top_F & \widetilde{f_1} = \top_F \vee \ldots \vee \widetilde{f_n} = \top_F \quad (6.10) \\ \bigcup_{\substack{f_i \in \widetilde{f_i} \\ E \in \mathcal{E}}} \{f' \mid \langle f', _\rangle \in [\![fop]\!](E, f_1, \ldots, f_n)\} & \text{o.w.} \quad (6.11) \end{cases}$$

Figure 6.8: Mostly-concrete semantic function. In $I_\top$, s is again the distinguished program start label.

both branches are taken. If any argument to the lifted version of $\widetilde{[\![fop]\!]}$ is unknown (i.e., $\top_F$) then the result is also unknown. Otherwise, the lifted version computes the result of applying the *fop* to every possible valuation of arguments in any environment. This enumeration can yield large (or even infinite sets), but the semantics of Figure 6.8 do not abstract these sets to the $\top_F$ value, which simplifies our formal presentation. In practice, our implementation falls back on $\top_F$ for infinite or unmanageably large sets of values (Section 6.7).

Like concrete interpretation, mostly-concrete interpretation can exploit application-specific knowledge and precisely model *fop*s that would otherwise be treated as non-deterministic. As described in Section 6.3.1, we model effectively deterministic *fop*s by introducing hypotheses on the domain of program environments $\mathcal{E}$. Thus, despite taking the union over $\mathcal{E}$, the definition of $\widetilde{[\![fop]\!]}$ uses only environment models consistent with application-specific information.

Unlike *fop*s, operations on values of types from $\mathcal{A}$ are modeled with imprecise, albeit sound, semantics. Mostly-concrete interpretation can support arbitrary abstractions of $\mathcal{A}$ values precisely because it makes no attempt to interpret *aop*s and therefore does not need to "understand" the abstraction domain $\widehat{A}$. However, Concerto does not suffer a precision loss from these coarse semantics; due to the state separation hypothesis, all *aop*s, bc_a and comparisons over $\mathcal{A}$ occur in statements labeled ℓ_a, which are modeled in Concerto with abstract interpretation. In other words, the mostly-concrete semantics for operations over $\mathcal{A}$ values are imprecise, but never actually executed in the mostly-concrete interpreter.

Theorem 4. $\alpha_F \circ F \sqsubseteq_{R \to \widetilde{R}} I_\top \circ \alpha_F$, *i.e.,* $I_\top$ *is a sound over-approximation of* F.

Proof Sketch. By case analysis on the definitions of $step^F$ and $step^\top$. The full proofs are available in Appendix B.1.1. □

$$\overline{R} = (\mathcal{L}_F \to \widetilde{S}) \times (\mathcal{L}_A \to \widehat{S}) \qquad \alpha_C(r) = \langle \lambda\vec{\ell} : \mathcal{L}_F.\alpha_F(r)[\vec{\ell}],\ \lambda\vec{\ell} : \mathcal{L}_A.\alpha_A(r)[\vec{\ell}] \rangle$$

$$\widetilde{\tau} : \widehat{S} \to \widetilde{S} \text{ such that: } \forall \vec{\ell}, r : R.\alpha_F(r)[\vec{\ell}] \sqsubseteq_{\widetilde{R}} \widetilde{\tau}(\alpha_A(r)[\vec{\ell}])$$

$$\widetilde{inj} : \overline{R} \to \widetilde{R} \text{ where: } \widetilde{inj}(\langle \widetilde{m}, \widehat{m} \rangle)[\vec{\ell}] = \begin{cases} \widetilde{m}[\vec{\ell}] & \vec{\ell} \in \mathcal{L}_F \\ \widetilde{\tau}(\widehat{m}[\vec{\ell}]) & \vec{\ell} \in \mathcal{L}_A \end{cases}$$

$$\widehat{\tau} : \widetilde{S} \to \widehat{S} \text{ such that: } \forall \vec{\ell}, r : R.\alpha_A(r)[\vec{\ell}] \sqsubseteq_{\widehat{S}} \widehat{\tau}(\alpha_F(r)[\vec{\ell}])$$

$$\widehat{inj} : \overline{R} \to \widehat{R} \text{ where: } \widehat{inj}(\langle \widetilde{m}, \widehat{m} \rangle)[\vec{\ell}] = \begin{cases} \widehat{m}[\vec{\ell}] & \vec{\ell} \in \mathcal{L}_A \\ \widehat{\tau}(\widetilde{m}[\vec{\ell}]) & \vec{\ell} \in \mathcal{L}_F \end{cases}$$

Figure 6.9: The combined mostly-concrete and abstract domain definitions, along with the domain transformers and derived injection functions.

6.4.3 *Combined Abstract and Mostly-Concrete Interpretation*

We now show how to combine the mostly-concrete and abstract interpreters. The approach broadly mirrors the strawman approach from Section 6.4.1. Specifically, combined interpretation operates over a combined domain $\overline{R}$. Like its strawman counterpart $\overline{R}_0$, $\overline{R}$ represents reaching states in the application with abstract state $\widehat{S}$, but uses mostly-concrete states $\widetilde{S}$ for the framework instead of $\mathbb{P}(S)$. CONCERTO's combined interpretation also injects the combined state representation into the "native" formats expected by the abstract and mostly-concrete interpreters. However, instead of using abstraction and concretization functions as in Section 6.4.1, we use *domain transformers* to soundly translate between state representations *without* requiring one of the abstract or mostly-concrete state representation to be more precise than the other.

The combined analysis domain $\overline{R}$ and an abstraction function α_C are defined in Figure 6.9. As α_A and α_F are complete join morphisms, α_C is itself a complete join morphism, and thus there exists some γ_C such that R and $\overline{R}$ form a Galois connection. The monotone functions $\widetilde{\tau}$ and $\widehat{\tau}$ in Figure 6.9 are the domain transformers described above:

$\widetilde{\tau}$ transforms a state from the abstract interpreter into a mostly-concrete state, and $\widehat{\tau}$ performs a transformation in the opposite direction. They are both functions provided by the analysis that must fulfill the following conditions:

$$\forall \vec{\ell}, r : R. \alpha_A(r)[\vec{\ell}] \sqsubseteq_{\widehat{S}} \widehat{\tau}(\alpha_F(r)[\vec{\ell}]) \quad (6.12) \qquad \forall \vec{\ell}, r : R. \alpha_F(r)[\vec{\ell}] \sqsubseteq_{\widetilde{S}} \widetilde{\tau}(\alpha_A(r)[\vec{\ell}]) \quad (6.13)$$

Intuitively, conditions (6.12) and (6.13) state that the transformers must be consistent with the target domain's abstraction function. As a consequence, (6.13) implies that any relational information present in $\widehat{R}$ must be discarded when moving to the non-relational domain $\widetilde{R}$. The $\widehat{A}$ values produced by α_F are the result of a non-relational abstraction function α_v, and by the inequality of Equation (6.13), the result of $\widetilde{\tau}$ can do no better. Despite this restriction on relational abstractions in the mostly-concrete domain, the above requirements on the the domain transformers do not provide information about the relative precision of the two domains. In fact, as mentioned above and illustrated below, it may not necessarily be the case that one of the domains is more precise than the other.

Example 6.4.3 (Trivial Transformers)**.** Consider the domain of Example 6.3.2 and the definition of α_v from Example 6.4.1. The two state representations are equal ($\widehat{S} = \widetilde{S}$), and $\widetilde{\tau} = \widehat{\tau} = id$. In other words, the two domains have the same expressive power. □

Example 6.4.4 (Relational Domain)**.** Suppose instead of representing integers with of the signedness domain used in Example 6.3.2, we used the Pentagon domain of Example 6.4.2 with the corresponding α_v and $\widehat{A} = Intv$. Then the abstract domain is $X_a \rightarrow Intv \times X_a \rightarrow \mathbb{P}(X_a) \times X_f \rightarrow \mathbb{P}(V_f)^\top$, where the first two components are respectively the interval environment and strict upper bounds of integers described in [130]. This abstract domain has used the $\mathbb{P}(V_f)^\top$ representation for framework variables. As with the embedding of $\widehat{A}$ in the mostly-concrete interpreter, this embedding of mostly-concrete values into the abstract domain is feasible due to the state separation hypothesis: i.e., *fop*s will not appear in code analyzed by the abstract interpreter.

The domain transformers may be defined as:

$$\widetilde{\tau}(\langle b, s, m\rangle) = \langle b, m\rangle \qquad \widehat{\tau}(\langle b, m\rangle) = \langle b, \lambda x : X_a.\emptyset, m\rangle$$

That is, $\widetilde{\tau}$ discards the relational information from $\widehat{S}$ when moving to $\widetilde{S}$, and $\widehat{\tau}$ uses the top element of the strict upper bound domain in the output (as the input mostly-concrete state does not have any relational information). In this example, the abstract domain is more precise, i.e. $\widehat{S} \underset{\widetilde{\tau}}{\overset{\widehat{\tau}}{\leftrightarrows}} \widetilde{S}$. □

Example 6.4.5 (Trivial Abstract Domain)**.** Consider a domain $X_a \to \mathbb{P}(\{-, 0, +\}) \times X_f \to \mathbf{1}$, where $\mathbf{1}$ is unary domain whose single element tt represents "any possible value", i.e., the analysis does not try to reason about maps, strings, or I/O. Then $\widetilde{\tau}(\langle m, z\rangle) = \langle m, \lambda x : X_f.\top_F\rangle$ and $\widehat{\tau}(\langle m, t\rangle) = \langle m, \lambda x : X_f.\mathtt{tt}\rangle$, and $\widetilde{S} \underset{\widehat{\tau}}{\overset{\widetilde{\tau}}{\leftrightarrows}} \widehat{S}$, i.e., the mostly-concrete domain is more precise than the abstract domain. □

Example 6.4.6 (Mixed Expressiveness)**.** Finally, consider a combination of Examples 6.4.4 and 6.4.5 where integers (i.e., variables in X_a) are modeled in the abstract domain with pentagons, $\widehat{A} = \mathit{Intv}$, and framework types (maps and strings) are modeled with the highly imprecise domain $\mathbf{1}$. Then $\widetilde{\tau}(\langle b, s, z\rangle) = \langle b, \lambda x : X_f.\top_F\rangle$ and $\widehat{\tau}(\langle b, m\rangle) = \langle b, \lambda x : X_a.\emptyset, \lambda x : X_f.\mathtt{tt}\rangle$. $\widetilde{\tau}$ and $\widehat{\tau}$ do not form a Galois connection: the abstract domain is more precise for integers, but the mostly-concrete domain more precisely represents maps and strings. □

We are ready to define the combined interpretation function $C : \overline{R} \to \overline{R}$ as follows:

$$C(X) = \left\langle I_\top \circ \widetilde{inj}(X)|_{\mathcal{L}_F},\ \widehat{F} \circ \widehat{inj}(X)|_{\mathcal{L}_A} \right\rangle \tag{6.14}$$

This definition closely mirrors Equation (6.1). The injection functions $\widetilde{inj}$ and $\widehat{inj}$ (defined in Figure 6.9) take the place of inj_R and $inj_{\widehat{R}}$ and translate the combined domain into $\widetilde{R}$ and $\widehat{R}$ by using the domain transformers $\widetilde{\tau}$ and $\widehat{\tau}$ respectively. The fully-concrete semantic function F has also been replaced with the mostly-concrete semantic function $I_\top$.

We can state the soundness of C according to α_C:

$$LB(r)[\ell^{\circ} \rightsquigarrow \ell^{\bullet}] = step^{LB}(\sqcup_{p \in pred(\ell)} r[p^{\bullet} \rightsquigarrow \ell^{\circ}], \ell) \sqcup \begin{cases} step^{LB}(\bot_{\widetilde{S}}, s) & \ell = s \\ \bot_{\widetilde{S}} & o.w. \end{cases}$$

$$LB(r)[\ell_f^{\bullet} \rightsquigarrow \ell^{\circ}] = r[\ell_f^{\circ} \rightsquigarrow \ell_f^{\bullet}] \qquad\qquad LB(r)[\ell_a^{\bullet} \rightsquigarrow \ell^{\circ}] = \bot$$

$$step^{LB}(\widetilde{in}, \ell) = \begin{cases} \widetilde{in} & prog[\ell] = \texttt{if} \ \ldots \ \vee prog[\ell] = \texttt{goto} \ \ell & (6.15) \\ \widetilde{in}[x \mapsto \widetilde{in}[y]] & prog[\ell] = x = y & (6.16) \\ \widetilde{in}[x \mapsto \bot_{\widehat{A}}] & prog[\ell] = x = bc_a \ \vee prog[\ell] = x = aop(v_1, \ldots, v_n) & (6.17) \\ \widetilde{in}[x \mapsto \top_F] & prog[\ell] = x = fop(v_1, \ldots, v_n) \vee prog[\ell] = x = bc_f & (6.18) \end{cases}$$

Figure 6.10: Semantic function defining a lower-bound on the precision of the abstract interpretation.

Theorem 5. $\alpha_C \circ F \sqsubseteq_{R \to \widetilde{R}} C \circ \alpha_C$

Proof Sketch. By Theorem 4, the assumed soundness of $\widehat{F}$, and from the fact that $\alpha_F \sqsubseteq \widetilde{inj} \circ \alpha_C$ and that $\alpha_A \sqsubseteq \widehat{inj} \circ \alpha_C$. □

As C is a monotone function on a complete lattice it has a least fixed point [180]. From Theorem 5 and [49] we then have that: $\alpha_C(\mathit{lfp}\, F) \sqsubseteq \mathit{lfp}\, C$.

6.4.4 *Conditions for Increased Precision*

C is sound, but may not necessarily be more precise than the original function $\widehat{F}$. We now discuss a set of sufficient conditions for when C is at least as (if not more) precise as $\widehat{F}$.

First, we define a function $LB : \widetilde{R} \to \widetilde{R}$ as shown in Figure 6.10. Intuitively, *LB* provides a lower-bound on the precision of the abstract semantic function; showing that $I_\top$ can "do better" than this lower bound will imply that $I_\top$ provides improved precision compared to $\widehat{F}$ on framework code. *LB* specifies an imprecise lower bound for modeling framework operations and comparisons, but provides no lower bound on the

precision for application operations or comparisons. However, as *LB* is non-relational, the definition implies that $\widehat{F}$ must also be non-relational.

C is more precise (as defined below) if the following conditions hold:

$$\widehat{\tau} \circ \widetilde{\tau} = \mathrm{id} \qquad LB \circ \dot{\widetilde{\tau}} \sqsubseteq_{\widehat{R} \to \widetilde{R}} \dot{\widetilde{\tau}} \circ \widehat{F} \qquad \forall U \subseteq \widehat{S}. \widetilde{\tau}(\sqcup U) = \sqcup_{\widehat{s} \in U} \widetilde{\tau}(\widehat{s}) \tag{6.19}$$

That is, if no precision is lost by moving to the domain $\widetilde{S}$ and then back to $\widehat{S}$, if *LB* is a lower bound on the precision of $\widehat{F}$, and if $\widetilde{\tau}$ is a complete join morphism. In the above, $\dot{\widetilde{\tau}}$ denotes the pointwise extension of $\widetilde{\tau}$.

As C and $\widehat{F}$ operate over different domains, we first introduce a function $\widehat{\mathrm{proj}} : \widehat{R} \to \overline{R}$ to *project* the $\widehat{R}$ domain into the combined domain $\overline{R}$:

$$\widehat{\mathrm{proj}}(\widehat{s}) = \langle \lambda \vec{\ell} : \mathcal{L}_F . \widetilde{\tau}(\widehat{s}[\vec{\ell}]),\ \lambda \vec{\ell} : \mathcal{L}_A . \widehat{s}[\vec{\ell}] \rangle$$

$\widehat{\mathrm{proj}}$ uses $\widetilde{\tau}$ to translate abstract to mostly-concrete states, reversing the $\widehat{\mathrm{inj}}$ operation.

If the above conditions hold, we can prove:

Theorem 6. $\widehat{\mathrm{inj}}(\mathit{lfp}\, C) \sqsubseteq_{\widehat{R}} \mathit{lfp}\, \widehat{F}$

Proof Sketch. From the assumptions in Equation (6.19) and by case analysis on $\mathit{step}^{\top}$ and $\mathit{step}^{\mathrm{LB}}$, it can be shown that $C \circ \widehat{\mathrm{proj}} \sqsubseteq_{\widehat{R} \to \overline{R}} \widehat{\mathrm{proj}} \circ \widehat{F}$ (Lemma 6 in appendix Appendix B.1.3), whence it follows by straightforward transfinite induction (Lemma 7 in B.1.3) that $\mathit{lfp}\, C \sqsubseteq_{\overline{R}} \widehat{\mathrm{proj}}(\mathit{lfp}\, \widehat{F})$. As $\widehat{\mathrm{inj}}$ is monotone, we have: $\widehat{\mathrm{inj}}(\mathit{lfp}\, C) \sqsubseteq_{\widehat{R}} \widehat{\mathrm{inj}} \circ \widehat{\mathrm{proj}}(\mathit{lfp}\, \widehat{F})$, whence we have $\widehat{\mathrm{inj}}(\mathit{lfp}\, C) \sqsubseteq_{\widehat{R}} \mathit{lfp}\, \widehat{F}$, as $\widehat{\mathrm{inj}} \circ \widehat{\mathrm{proj}} = \mathrm{id}$ by Equation (6.19). □

Theorem 6 states that the approximation of reaching states computed by the combined interpretation function C is at least as precise as that computed by the abstract interpreter. The inequality is *not* strict: whether CONCERTO matches or exceeds the precision of plain abstract interpretation depends on the program and the abstract semantics and domain. As a trivial example, on a program with only application statements (i.e., with labels drawn from ℓ_a) CONCERTO will necessarily do no better than plain abstract interpretation.

Example 6.4.7 (Signedness Analysis)**.** The signedness example of Section 6.2 satisfies the above conditions, and thus combined interpretation with Concerto is at least as precise as plain abstract interpretation. As both $\widetilde{\tau}$ and $\widehat{\tau}$ are the identity function, parts 1 and 3 of Equation (6.19) are trivially satisfied. The abstract semantic function in Figure 6.7 uses very coarse approximations of *fop* operations, and therefore part 2 of Equation (6.19) is also satisfied. □

6.5 *Procedures*

Our formalism so far does not support procedures, but they are essential. Section 6.5.1 sketches the addition of procedures to our formal language and extensions to the concrete and mostly-concrete semantics. Section 6.5.2 then demonstrates that by restricting control transfers between the framework and application to procedures calls and returns, Concerto does not need AI developers to provide the full domain transformers ($\widetilde{\tau}$ and $\widehat{\tau}$) introduced in Section 6.4.3 since it suffices to transform only parameter and return values across procedure boundaries. We delay until Section 6.8 how to extend our support for procedures to objects and methods in the style of Java.

6.5.1 *Interprocedural Semantics*

We assume a program now has the following form, where f ranges over procedure names:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{prog} & ::= & D* & \mathit{astmt} & ::= & \ldots \mid \texttt{return}\ x \mid x = f(y) \\ D & ::= & \texttt{proc}\ f(p)\{\ (\mathit{fstmt}^{\ell} \mid \mathit{astmt}^{\ell})^{*}\} & \mathit{fstmt} & ::= & \ldots \mid \texttt{return}\ x \mid x = f(y) \end{array}$$

We assume that procedure names are distinct, that there is a distinguished main procedure, and that the distinguished program start label s is the first statement of this procedure.

For a procedure call statement labeled with ℓ, we assume there are two pseudo-labels: ℓ_c and ℓ_r, corresponding to the point in program execution immediately before invoking

the function and immediately after the called function returns. We extend the *succ* and *pred* relations to include these pseudo-labels. For a call-site labeled ℓ, if control may flow from label ℓ' to ℓ, $(\ell', \ell_c) \in succ$, and similarly for (ℓ_r, ℓ') and control-flow from ℓ to ℓ'. In addition, the entry point label of the callee at ℓ is a successor of ℓ_c and the predecessor of ℓ_r is the return site label in the callee.

We extend the state definition to track the runtime stack: $R = \mathcal{L} \to \mathbb{P}(S^* \times \ell^* \times \mathcal{E})$, where S^* is a sequence of states S as defined in Section 6.3.2, and ℓ^* is a sequence of statement labels. Intuitively, these two components represent the runtime stack and the return locations for active procedure invocations respectively. We also assume that the domain of variables X includes a special return slot ρ.

We extend the definitions of F and $I_\top$ given in Sections 6.3.2 and 6.4.2 respectively as shown in Figure 6.11 to handle these new statement forms and flow edges. We omit the updated definitions of F and $I_\top$ at the distinguished start label; they are the obvious extensions to the terms in Figures 6.6 and 6.8. We use ρ to store the return value of functions with return type $\mathcal{F}$ and $\mathcal{A}$: we shall assume that there are two different versions of ρ of the appropriate type. Finally, $\mathcal{V}$ is now defined as: $\mathcal{V}(\mathcal{S}, x) = \{s[x] \mid \langle \vec{s_0} \circ s, _, _\rangle \in \mathcal{S} \wedge s[x] \neq \bot_V\}$.

We have proved (see Appendix B.2) that this updated definition of $I_\top$ is sound with respect to the updated definition of F. If the abstract interpretation is also sound with respect to this updated definition, and the $\widetilde{\tau}$ and $\widehat{\tau}$ functions still fulfill the conditions in Section 6.4 then all the results of the previous section still hold with no modifications to our formalisms or proofs.

6.5.2 *Interprocedural Domain Transformers*

We have so far treated procedure call and return as orthogonal to control-flow transfer between the application and framework, i.e., procedure bodies could be a mix ℓ_f and ℓ_a statements. In practice, we require all transfers between the application and framework

$$
\begin{aligned}
F(r)[\ell_r^\circ \rightsquigarrow \ell_r^\bullet] &= \bigsqcup_{p \in pred(\ell_r)} \{\langle s \circ s_r[x \mapsto s_c[\rho]], \mathcal{R}, E\rangle \mid \\
&\qquad \langle s \circ s_r \circ s_c, \mathcal{R}, E\rangle \in r[p^\bullet \rightsquigarrow \ell_r^\circ] \wedge \langle s \circ s_r, \mathcal{R}, _\rangle \in r[\ell_c^\circ \rightsquigarrow \ell_c^\bullet]\} \\
F(r)[p^\bullet \rightsquigarrow \ell_r^\circ] &= \{\langle s, \mathcal{R}, E\rangle \mid \langle s, \mathcal{R} \circ \ell, E\rangle \in r[p]\} \\
F(r)[\ell_c^\bullet \rightsquigarrow \ell'^\circ] &= \{\langle s \circ s_r \circ \iota_S[p \mapsto s_r[y]], \mathcal{R} \circ \ell, E\rangle \mid \langle s \circ s_r, \mathcal{R}, E\rangle \in r[\ell_c]\} \\
F(r)[\ell_c^\circ \rightsquigarrow \ell_c^\bullet] &= \bigsqcup_{p \in pred(\ell_c)} r[p^\bullet \rightsquigarrow \ell_c^\circ] \\
\mathit{step}^F(in, \ell) &= \begin{cases} in[\rho \mapsto in[x]] & \mathit{prog}[\ell] = \texttt{return}\ x \\ \mathit{as\ before} & o.w. \end{cases} \\
I_\top(\widetilde{r})[\ell_r^\circ \rightsquigarrow \ell_r^\bullet] &= \bigsqcup_{p \in pred(\ell_r)} \widetilde{r}[\ell_c][x \mapsto \widetilde{r}[p^\bullet \rightsquigarrow \ell_r^\circ][\rho]] \\
I_\top(\widetilde{r})[p^\bullet \rightsquigarrow \ell_r^\circ] &= \widetilde{r}[p] \\
I_\top(\widetilde{r})[\ell_c^\bullet \rightsquigarrow \ell'^\circ] &= [p \mapsto \widetilde{r}[\ell_c][y]] \\
I_\top(\widetilde{r})[\ell_c^\circ \rightsquigarrow \ell_c^\bullet] &= \bigsqcup_{p \in pred(\ell_c)} \widetilde{r}[p^\bullet \rightsquigarrow \ell_c^\circ] \\
\mathit{step}^\top(\widetilde{in}, \ell) &= \begin{cases} \widetilde{in}[\rho \mapsto \widetilde{in}[x]] & \mathit{prog}[\ell] = \texttt{return}\ x \\ \mathit{as\ before} & o.w. \end{cases}
\end{aligned}
$$

Figure 6.11: Extensions to the semantic functions F and $I_\top$ to support procedures. In the above definitions, p is the name of the parameter of the called procedures, y is the variable passed as the argument, and x is the variable in the caller to which the result of the function is defined. ι_S is a state where all variables are bound to $\bot_V$.

occur at procedure boundaries. In other words, each procedure may only contain ℓ_a statements or ℓ_f statements. This restriction is entirely reasonable: real frameworks encapsulate their functionality in methods/classes/modules/etc.: source programs would not mix framework and application code in the same procedure.

This (non-)restriction also significantly simplifies how CONCERTO transfers values between abstract and mostly-concrete interpretation. Recall that CONCERTO applies the domain transformers at transitions between the application and framework. However, the above syntactic restriction implies that CONCERTO needs the domain transformers only at procedure entry and exit. Further, at procedure entry, the mostly-concrete interpreter can access only the program state reachable via the parameters. Thus, instead of using $\widetilde{\tau}$ at application-to-framework calls, the AI may directly provide mostly-concrete arguments to the mostly-concrete component, which then binds the arguments in a empty local state. Similarly, we assume that when given mostly-concrete arguments the abstract interpreter can construct a sound abstract procedure entry state. (This assumption is possible because our language is statically scoped and has no global variables, heaps are discussed in Section 6.8.) As a consequence, instead of using $\widehat{\tau}$ at framework-to-application calls, the mostly-concrete interpreter may provide mostly-concrete arguments to the AI, which binds them in an abstract empty state, transforming the mostly-concrete values into a "native" abstract representation as necessary.

CONCERTO uses a similar process for return flows. The mostly-concrete semantics use the exit state of a called procedure only to extract the procedure's return value. Instead of using $\widetilde{\tau}$ at return flows from the application to framework, the AI simply provides a mostly-concrete representation of the return slot ρ. We likewise assume that the AI is interested only in the value of ρ in callee exit states. Thus, at framework-to-application return flows, the mostly-concrete interpreter may provide only the mostly-concrete return value, which the AI transforms into a native representation as necessary.

The direct exchange of values sketched above obviates implementations of $\widetilde{\tau}$ and $\widehat{\tau}$, but to retain soundness, the values exchanged must be consistent with those produced

by some $\widetilde{\tau}$ and $\widehat{\tau}$ that satisfy the definitions given in Section 6.4.3. For example, consider an application-to-framework call where the abstract entry state is $\widehat{s}$. The mostly-concrete argument provided to the mostly-concrete interpreter must therefore be $\widetilde{\tau}(\widehat{s})[p]$ for some $\widetilde{\tau}$, and where p is the parameter of the called procedure. Thus, any valid domain transformers $\widetilde{\tau}$ and $\widehat{\tau}$ provide correctness specifications for the exchange of values. However, a trivial way to ensure soundness is to generate complete definitions for some $\widetilde{\tau}$ and $\widehat{\tau}$, and hand-simplify the results of applying the transformers at procedure call and return. For example, to correctly generate mostly-concrete return values at application-to-framework return flows, it is sufficient to use a simplification of $\lambda\widehat{s}.\widetilde{\tau}(\widehat{s})[\rho]$.

Context Sensitivity The above discussion does not treat context-sensitivity in the abstract interpretation. In our implementation we have the AI provide functions to compute the callee context at framework-to-application calls (Section 6.8.3). Further the mostly-concrete semantics in Figure 6.11 are context-*insensitive*, which is not very precise. In practice, we unroll the call-graph up to recursive cycles in the mostly-concrete interpreter, effectively giving unlimited context-sensitivity. We discuss these two techniques in Section 6.8.3. We have not formalized our approach to context-sensitivity, although adapting our proofs and formalism is a straightforward, albeit tedious, extension.

6.6 Iteration Strategy

We now briefly describe the iteration strategy used by Concerto. Our implementation runs mostly-concrete and abstract interpreters in parallel until they converge to a local fixed-point on their respective partitions of the program (framework and application code respectively). The results are then exchanged between the two interpreters (under the syntactic restriction of Section 6.5.2, this exchange is performed at procedure boundaries as described above) and the process repeats until the overall process converges to a fixed-point. We refer to this process as *subfixpoint iteration*.

The subfixpoint iteration scheme sketched above is very similar to chaotic asyn-

chronous iteration with memory [46], with one key difference. Before starting iteration in the mostly-concrete component, subfixpoint iteration discards results at framework statements computed in previous rounds of subfixpoint iteration. In other words, after exchanging information with the abstract interpretation component, the mostly-concrete component iterates a "fresh" mostly-concrete interpreter, seeded with only information received from the abstract interpreter. We have shown in Appendix B.3 that this process converges to the least fixed point of C. Our proof depends on the following property of subfixpoint iteration (Lemma 21 in the appendix): any information discarded between runs of the mostly-concrete interpreter can be soundly recovered with enough iterations in the next run of the mostly-concrete interpreter.

This iteration strategy justifies analyzing application-to-framework calls by spawning a fresh mostly-concrete interpreter seeded with mostly-concrete arguments that flow into the framework from the application provided by the AI. We describe this process in more detail in Section 6.8.

6.7 *Widening and Finitization*

Two significant challenges remain to a realizable implementation. First, although we have proved that subfixpoint iteration converges to the least fixed point of C, it may not do so in finite time; the domain $\overline{R}$ does not possess an ascending chain condition that will ensure convergence in finite steps. Second, we have not yet guaranteed that mostly-concrete interpretation manipulates only finite sets of values. To address the first issue, we apply widening [47] during iteration. We address the second issue by forbidding infinite sets of values, and describe how the mostly-concrete interpreter uses $\top_F$ in practice to avoid materializing infinite sets.

Widening Following the vocabulary of [27], we require two *widening point sets* $\mathcal{W}_A$ and $\mathcal{W}_F$ for application and framework statements, respectively. A widening point set is a set of statement labels such that, if during iteration the states at all widening points stabilize,

then the overall iteration stabilizes in a finite number of steps. We further require that if the states at all widening points in two iteration sequences stabilize to the same set of values, then the two sequences stabilize to the same result. We leave the choice of $\mathcal{W}_A$ up to the abstract interpretation, although we expect most interpreters will use a variation on the strategy described by [27]. In our mostly-concrete interpreter, we use the headers of unbounded loops and the entry point of a representative method selected from recursive cycles (including sub-cycles).

We assume that the analysis provides widening operators $\triangledown_{\widehat{A}}$ for values of type $\widehat{A}$ and $\widehat{\triangledown}$ for abstract states $\widehat{S}$. From $\triangledown_{\widehat{A}}$, we derive a widening operator for mostly-concrete states:

$$\langle m_f, m_a \rangle \widetilde{\triangledown} \langle m_f', m_a' \rangle = \Big\langle \Big(\lambda x : X_f. \begin{cases} m_f & m_f'[x] \sqsubseteq m_f[x] \\ \top_F & o.w. \end{cases}\Big),\ \big(\lambda x : X_a. m_a[x] \triangledown_{\widehat{A}} m_a'[x]\big) \Big\rangle$$

Given the above assumptions and definitions, we ensure termination as follows. We again iterate the abstract and mostly-concrete interpreters in parallel, except we instrument the abstract semantic function $\widehat{F}$ and $I_\top$ to apply widening operations at the locations in the widening point sets $\mathcal{W}_A$ and $\mathcal{W}_F$ respectively. We have proved (Appendix B.4) that these individual iterations terminate in a finite number of steps. After the two interpreters stabilize, they exchange results and the process repeats; with the mostly-concrete interpreter again discarding previously computed information as described above. This process stabilizes in a finite number of steps to an over-approximation of *lfp* C (see Appendix B.4.1).

Precision Section 6.4.4 gave conditions for when *lfp* C will be at least as precise as *lfp* $\widehat{F}$. Whether this precision result also translates to the widened subfixpoint iteration presented above will depend on the choice of widening operators. As widening operators are not necessarily monotone, the instrumented $\widehat{F}$ and $I_\top$ functions are not necessarily monotone either. Without monotonicity, reasoning about the relative precision of subfixpoint iteration with widening is difficult. This result is not surprising; as noted in

[50], when using widening the order of iteration can have a significant impact on the precision of the final result.

Finitization Finally, to ensure Concerto manipulates only finite sets, we require that the AI does not provide infinite argument or return value representations to the mostly-concrete interpreter. We also extend the definition of $I_\top$ to finitize the result of all *fop* operations. Whenever applying *fop* to two arguments would produce an infinite set of values (or an otherwise impractically large set, e.g., all 32-bit machine integers) the modified semantic function abstracts this set with $\top_F$. This finitization is sound (see Comment 8.1 in Appendix B.1.1), while avoiding materializing infinite sets in our implementation.

6.8 Extensions for a Realistic Prototype

We have implemented a prototype combined interpreter for a subset of Java.[4] Our subset includes 1. interfaces and dynamic dispatch, 2. reflection, 3. dynamically sized arrays, and 4. statically unbounded loops. We include two primitives to simulate I/O. The first, `read()`, reads an integer from a deterministic stream. This primitive models reading from a deterministic configuration file. The second primitive, `nondet()` reads from a nondeterministic stream, which simulates, e.g., packets received from the network or user input. While falling considerably short of the full complexity of Java, we can use these language features to effectively simulate some of the most difficult to analyze code idioms we have encountered in real-world framework implementations (see Section 6.9).

Our prototype takes as input an abstract interpretation implementation which exposes basic operations required by Concerto. We introduce these operations incrementally as we extend our basic procedural language to support objects, methods, etc.

[4]Our implementation prototype is open-source and is available at `https://github.com/uwplse/concerto`.

6.8.1 Objects and the Heap

We first consider only class-based objects with fields, deferring methods to Section 6.8.2 and primitives, interfaces, and libraries to Section 6.8.4. We assume each concrete class belongs either to the framework or the application, taking $\mathcal{F}$ to be the framework classes and $\mathcal{A}$ the application classes. We also extend our concrete state to include a concrete heap: $S = \mathcal{E} \times (X \to V) \times H$. Object allocation and field manipulation are defined via *aop*s and *fop*s that additionally side-effect the heap (we leave a full formalization to future work). This formulation implies that framework code may not directly manipulate the object fields of application classes and vice versa. However, as argued in Section 6.2, real-world applications and frameworks almost exclusively communicate via functional interfaces.

By classifying each class as either framework or application, we can effectively partition the program's runtime heap H into H_a containing application objects and H_f containing framework objects, i.e., $S = \mathcal{E} \times (X \to V) \times H_a \times H_f$. We can then use different abstractions for H_a and H_f.

In the mostly-concrete interpreter, object operations on framework classes manipulate mostly-concrete heaps of type $\widetilde{H_f}$.[5] $\widetilde{H_f}$ is equipped with join, ordering, and widening operators. The mostly-concrete interpreter does not use its own abstraction for H_a, using one provided by the abstract interpreter (see below). Similarly, in the abstract interpreter, object operations on application classes operate on an abstract representation of H_a, i.e., the application heap component.[6] Concerto does *not* make any assumptions on the *internal* heap representation used by the abstract interpretation. However, the abstract interpretation must provide two functions: $\texttt{projectH} : \widehat{S} \to \widehat{H_a}$ and $\texttt{injectH} : \widehat{H_a} \to \widehat{S} \to \widehat{S}$ such that $\forall \widehat{s}. \widehat{s} \sqsubseteq_{\widehat{S}} \texttt{injectH}\ (\texttt{projectH}\ \widehat{s})\ \widehat{s}$ for some type $\widehat{H_a}$ defined by the analysis. The

[5] We do not use a fully concrete heap to handle object allocations in unbounded loops. Our widening operator detects such cases and introduces mostly-concrete *summary objects* where appropriate.

[6] Technically, the abstract interpretation may also operate on an abstraction of H_f. However, all object operations that mutate H_f are modeled by the mostly-concrete interpreter, so in practice the abstract interpreter only uses an abstraction of H_a.

AI must provide widening, join, and ordering operations for $\widehat{H_a}$.

To tie these two heap representations together, we extend the mostly-concrete state representation to include mostly-concrete and abstract heaps: $\widetilde{S} = (X_f \to \mathbb{P}(V_f)^\top) \times (X_a \to \widehat{A}) \times \widetilde{H_f} \times \widehat{H_a}$. V_f is the domain of mostly-concrete heap locations, and $\widehat{A}$ is an abstract representation of objects of type $\mathcal{A}$. During execution, the mostly-concrete interpreter updates the mostly-concrete heap and threads the abstract heap representation through unchanged. Concerto requires that the abstract interpreter also threads the mostly-concrete heap through its interpretation. To ensure the concrete heap is correctly handled, the abstract interpretation must operate over an instrumented state representation, $\widehat{S} \times H_f$. We provide APIs for the AI to manipulate this instrumented representation.

The approach described so far does not allow for framework objects to store references to application objects and vice versa. To relax this restriction, we require the abstract interpretation meets some additional conditions. First, the abstract heap must represent fields and variables with type $\mathcal{F}$ with the domain $\mathbb{P}(V_f)^\top$. Next, when performing a write of $v' : \mathbb{P}(V_f)^\top$ to a field/variable of type $\mathcal{F}$ with the existing value $v : \mathbb{P}(V_f)^\top$, the value v'' of the field/variable after the write must satisfy the constraint $v'' \sqsubseteq v' \sqcup v$, i.e., the new value is bounded above by the result from a weak update. This requirement ensures that the abstract interpretation never produces mostly-concrete object locations "out of thin air" that may not have yet been allocated in the mostly-concrete heap component. A similar concern exists when storing values of type $\mathcal{A}$ into the concrete heap: any abstractions stored into the concrete heap must remain valid, and updates via aliasing should be propagated to these values. A sufficient condition is for the abstract interpreter to internally use an abstract heap $\widehat{Loc} \rightharpoonup \widehat{\mathcal{O}}$ where $\widehat{Loc}$ is a finite domain of abstract locations and $\widehat{\mathcal{O}}$ are abstract objects, and to take $\widehat{A} = \mathbb{P}(\widehat{Loc})$ and $\widehat{H_a} = \widehat{Loc} \rightharpoonup \widehat{\mathcal{O}}$.

6.8.2 *Methods and Domain Transformers*

We require that methods in application classes contain only application code, and similarly for framework classes. As described in Section 6.5.2, this (non-)restriction ensures that values change representation only at method boundaries.

For framework-to-application calls, the abstract interpreter must expose a method `interpret` that analyzes a method m in context C (see below), with mostly-concrete arguments $a_1, a_2, \ldots, a_n$, and abstract receiver $\widehat{r}$. When Concerto encounters a method call in the mostly-concrete interpreter with a base pointer $\widehat{r}$ of type $\widehat{A}$ (i.e., static type $\mathcal{A}$), it yields into the abstract interpreter by passing the mostly-concrete arguments, abstract receiver, and a computed context C to `interpret`. `interpret` is responsible for constructing an initial abstract state for m and then performing abstract interpretation over the method body. When analysis of m is complete, `interpret` returns a mostly-concrete representation of the return value. This process mirrors the one described in Section 6.5.2. However, Concerto additionally instruments the above process to inject the caller's mostly-concrete and abstract heaps into the callee abstract state, and similarly extract the abstract and mostly-concrete heaps from the abstract exit state.

Concerto also provides an API for the AI to yield into the mostly-concrete interpreter when it encounters a call back into the framework. The AI calls this API method with a mostly-concrete receiver and arguments as well as the abstract caller state. The mostly-concrete interpreter extracts the two heaps from this caller state, binds the argument and receiver values, and begins executing the called method. When execution of the method completes, Concerto injects the resulting mostly-concrete and abstract heaps into the provided abstract caller state, and returns this updated state and a mostly-concrete return value to the AI.

6.8.3 Context-Sensitivity and Mostly-Concrete Interpretation

Concerto supports context-sensitive analyses. Our implementation is polymorphic over the type of contexts, leaving the representation entirely to the client abstract interpretation. When computing the context for an application method call, Concerto passes information about the current state, call site, and call stack to a `mkContext` method exposed by the AI. `mkContext` is responsible for computing the analysis context C for the method call and returning it to Concerto, which passes the computed context to the `interpret` function as described above.

At application-to-framework calls, Concerto spawns a fresh mostly-concrete interpreter and runs it until the called method returns. Where possible, this interpreter unrolls all statically bounded loops and unfolds the call-graph (effectively giving unlimited context-sensitivity and some path-sensitivity). However, due to nondeterministic program inputs or imprecision in the abstract interpreter, mostly-concrete interpretation may encounter nondeterministic conditionals, loops, and unbounded recursive cycles.

To handle nondeterministic choice, Concerto could fork two interpreters and continue execution down each path in parallel as in Klee [35] or Java PathFinder [30]. While sound, this approach would encounter the exponential explosion of paths common in symbolic execution. Instead, Concerto forks two interpreters at nondeterministic branches and executes both branches in parallel up to the conditional control-flow join point. At the control-flow join point, the two interpreter's states are joined and execution continues along a single thread of execution.

To ensure termination in the presence of nondeterministic loops or unbounded recursion, the mostly-concrete interpreter is instrumented to detect potentially infinite loop unrolling or call-graph unfolding and then falls back on over-approximation using widening.

6.8.4 Allowing State Separation Violations

The state separation hypothesis applies to the language presented thus far; the choice of whether a variable is modeled by an abstract value or mostly-concrete value could be made based on types, and operations on abstract values can only occur in the abstract interpreter and similarly for mostly-concrete values. However, the state separation hypothesis is violated if we add primitive types, interfaces, and common library types such as `Hashtable` to our supported subset of Java; e.g., frameworks may interrogate or modify an integer produced by application code. Although we could restrict the use of primitives and libraries to only application or framework code, and further require that all implementers of an interface must be either application or framework classes, such a restriction would be unrealistic. We therefore describe how to handle these features as well as direct mutation of application objects by the framework and vice versa.

Primitive Types and Operations

Our limited subset of Java supports only integers with basic arithmetic operations and comparisons. Integer values are represented in the mostly-concrete interpreter with a sum type: $\mathbb{P}(\mathbb{N})^{\top} + \widehat{A}$. When an integer abstraction of type $\widehat{A}$ flows into a framework method we automatically lift it into the sum type. We use $\widehat{A}$ as the abstract representation of objects and integers: in practice we expect that internally $\widehat{A}$ is a union of objects and integers abstractions.

When executing an arithmetic or comparison operation, CONCERTO checks if both operands are of type $\mathbb{P}(\mathbb{N})^{\top}$. If so, then the interpreter executes the concrete arithmetic operation lifted to the $\mathbb{P}(\mathbb{N})^{\top}$ domain. If one or both of the operands are of type $\widehat{A}$, then CONCERTO uses a $\texttt{lift} : \mathbb{P}(\mathbb{N})^{\top} \to \widehat{A}$ method exported by the abstract interpreter to convert the powerset representation into an $\widehat{A}$ representation. A sound choice for such a method is to simply return the top element for integers in $\widehat{A}$. After conversion, CONCERTO calls methods exposed by the abstract interpretation that perform the primitive

operations on elements of $\widehat{A}$, e.g. `plus` $: \widehat{A} \to \widehat{A} \to \widehat{A}$, `minus` $: \widehat{A} \to \widehat{A} \to \widehat{A}$, etc. As with the lift operation, a sound choice for these functions is to return the maximal element for integers in $\widehat{A}$. Finally, in cases such as array indexing, Concerto may need to transform $\mathbb{P}(\mathbb{N})^\top + \widehat{A}$ into $\mathbb{P}(\mathbb{N})^\top$. The abstract interpretation must also expose a function `lower` $: \widehat{A} \to \mathbb{P}(\mathbb{N})^\top$. A sound choice for this function is to simply return $\top_F$.

At calls from the framework into the application, Concerto passes the sum representation directly to the abstract interpretation which may lift this sum type into a native representation. In practice, abstract interpretations gain precision by using the sum representation for integers, and lifting to a native representation on demand for arithmetic and comparison operations.

Interfaces and Library Types

An interface I may have implementers in the application and the framework. Thus, given a variable/field of type I, it may be statically unknown whether that variable/field contains an instance of a framework or application class. We resolve this ambiguity by requiring that the abstract interpretation and mostly-concrete interpreter use a combined object representation $\mathbb{P}(V_f) \times \widehat{A}$ for values with an interface type. Intuitively, $\langle v_f, \widehat{a} \rangle$: $\mathbb{P}(V_f)^\top \times \widehat{A}$ represents *either* a framework object that is abstracted by v_f *or* an application object abstracted by $\widehat{a}$. In the case that one of the components is the least element in its respective lattice then the interpretation of combined value is the interpretation of the non-bottom component. In principle, we could have used this product representation for primitive types, but found that in practice the sum type representation is easier to use.

When the mostly-concrete interpreter encounters a method call on an interface, it splits the receiver into its two components, and then performs a concrete method call on the concrete component while simultaneously performing the abstract method call by yielding into the abstract interpreter. The results from both method calls are then merged using the appropriate join operations and execution continues. Concerto exports an

API that performs the symmetric operation for interface calls encountered in application code by the abstract interpreter.

Both framework and application code may use the same standard libraries, breaking our simplifying assumption that the types used in the application and framework are disjoint. For example, Java types like `ArrayList`, `HashMap`, etc. are ubiquitous. Concerto supports these *library types* using the same product representation used for interfaces. Unlike interfaces, the choice of whether to model library objects using a mostly-concrete or abstract representation is not based on static type information, but on the allocation site. For example, an `ArrayList` allocated in the application will be modeled using abstract values, whereas an `ArrayList` allocated in the framework will be modeled mostly-concretely. This approach is possible because we assume that library types, like interfaces, only export object-oriented interfaces and Concerto does not need to support framework code that directly modifies the internal state of a library objects allocated in the application and vice versa.

Direct Field Access

We have so far assumed that the framework never directly accesses or mutates application object fields and vice-versa. As argued in Section 6.2, we expect this assumption holds for the vast majority framework-based applications. However, our approach can still be applied in the cases where the assumption does not hold, albeit with some precision penalty. We describe how to extend our implementation to support framework code that reads and writes application object fields; the approach for the application code directly accessing framework fields is symmetric.

We first consider the case where framework code reads an application object field. Recall that we model operations on application fields as *aop*s and that we use an extremely coarse model for *aop*s in our mostly-concrete semantics. Thus, we can soundly model reads of application object fields as simply returning the maximal element from the ap-

propriate lattice. For example, the mostly-concrete interpreter may use $\top_{\widehat{A}}$ to model the value read from a field of application type. Any future operations on this read value will necessarily be imprecise; the exact extent of this imprecision will depend on how the read value is used by the program.

Our approach for handling direct mutations of application fields is broadly similar. As field writes are modeled as *aop*s that side effect the abstract heap, mutations are coarsely modeled by simply havocing the abstract heap, i.e. the interpretation of a field write returns $\top_{\widehat{H_a}}$, where $\widehat{H_a}$ is the domain of abstract heaps. This coarse approach will cause greater imprecision compared to the field read case above, but we contend that direct field mutations are exceedingly rare in practice. As frameworks and applications are developed independently from one another, the framework implementation cannot guarantee any application-specific object invariants are preserved by a field mutation. A similar argument applies for direct mutations of framework field by the application code. Thus, we expect any imprecision introduced by this coarse modeling to be limited for real-world applications.

6.9 Evaluation

To evaluate the feasibility and benefits of our combined analysis approach, we implemented a small "web application framework" called Yawn (Your Analysis' Worst Nightmare) in the subset of Java supported by our prototype implementation. Yawn implements an accept loop which parses requests received on our language's nondeterministic I/O stream and routes these requests to application defined handlers. Yawn contains several difficult-to-analyze features found in real-world frameworks, including dependency injection, an embedded Lisp interpreter, and indirect flow. The dependency injection component and the Lisp interpreter heavily used reflection, and the run-time behavior of all three features is determined by the contents of a configuration file.

We implemented a simple application using the Yawn framework. The application's primary functionality is implemented as a collection of request handlers which perform

simple mathematical operations (e.g., summing two integers) on request parameters. The application uses in-memory state and a simulated database layer implemented as standalone modules. These handlers and modules are constructed and wired together using Yawn's dependency injection mechanism. Yawn also includes a filtering mechanism to preprocess requests. Our application applies a filter that uses Yawn's embedded Lisp interpreter to run a filtering program specified in the application's configuration file.

Next, we implemented three abstract interpreters that use different abstract domains, heap representations, and context sensitivity. These analyses are summarized in Table 6.1. Pta performs VTA-style [178] call-graph construction using a type-based heap where abstract addresses are sets of type names. IFlow is an information flow integrity analysis [62] to find flows from untrusted sources to sensitive sinks. It uses the caller method as the context when analyzing a callee. For the heap abstraction, IFlow reuses the type-based heap from the Pta interpreter, and extends the reaching type domain with k-limited access paths [63, 100] that track which heap locations are tainted. Finally, the most complex (and expensive) analysis is Abc, an array bounds checker. Abc uses call site 1-CFA for contexts, and uses an abstract heap that maps abstract locations to abstract objects. An abstract location is pair consisting of an allocation site and the context in which the allocation occurred. Object values are abstracted by powersets of abstract locations. Integers are abstracted with an approximation of the reduced product [49] of the Interval domain *Intv* and inequalities between access paths, giving a weakly relational Pentagon domain [130]. As the choice of $\widehat{A}$ must be non-relational, the abstract representation of integers in $\widehat{A}$ is simply *Intv*. Abc also propagates inequalities induced by comparison operators making it partially path sensitive.

Finally, we ran each interpreter over the application twice: once using Concerto and once with standard abstract interpretation. Each run had a one hour time budget, and we measured the total time of each run. After each run, we collected call-graph information (Pta) or any alarms reported (Abc & IFlow). In the event of a timeout, we collected any information computed up to that point.

Table 6.1: Summary of abstract interpretations. **CS** is the context-sensitivity of the analysis if applicable. **PS** indicates if the analysis is path-sensitive.

Name	**CS?**	**Heap**	**Domain**	**Relational?**	**PS?**
PTA	No	Type-based	Reaching Types	No	No
IFLOW	Caller Method	Type-based	Access Paths/Reaching Types	No	No
ABC	Call site 1-CFA	Abstract Location	Pentagons/Abstract Locations	Yes	Yes

Table 6.2: Number of reports issued and execution times of the interpreters with (**Conc.**) and without (**Std. AI**) CONCERTO. t/o indicates a timeout. For a discussion of the PTA results, see the main text.

Analysis	**Time (Conc.)**	**Time (Std. AI)**	**Reports (Conc.)**	**Reports (Std. AI)**
PTA	4.7 s	1282.7 s	—	—
ABC	8.8 s	t/o	0	2
IFLOW	4.6 s	t/o	3	7

For every analysis, Concerto vastly outperformed plain abstract interpretation as shown in Table 6.2. Under plain abstract interpretation, Abc and IFlow timed out while Pta took approximately 275× longer than combined interpretation. The Abc and IFlow timeouts were caused by enormous strongly connected components due to sound but imprecise modeling of Yawn's use of reflection and indirection. Even with widening, propagating information through these cycles overwhelmed the abstract interpreters. Pta also encountered large strongly connected components, but the lack of context-sensitivity and simplicity of the abstract domains mitigated the performance impacts.

Further, the quality of analysis results was significantly worse with plain interpretation compared to Concerto. To evaluate the precision of Abc and IFlow, we classified the alarms reported as either true or false positives. Combined interpretation correctly found all 3 information leaks in our test application and also successfully verified that the application was free of out-of-bounds array accesses. In contrast, plain abstract interpretation reported 7 leaks and and 2 out-of-bounds accesses, respectively. For the array bounds checker, all these reports were false positives, and all but 3 were false positives for the information flow analysis. Additionally, as these results were collected after timeouts, they represent a lower bound on the imprecision of Abc and IFlow.

Pta does not find bugs, but produces a call-graph for a downstream analysis; we include it in our experiments to demonstrate the impact of combined interpretation on resolving reflective invocations in framework code. As a representative example, under plain interpretation Pta resolved the reflective allocations in the dependency injection facility to 38 possible types, compared to just 15 types under combined interpretation. Similarly, within the Lisp interpreter, plain interpretation resolved the reflective invocations to as many as 30 and no fewer than 8 callees, whereas combined interpretation resolved every invocation to a single callee.

6.10 Conclusion

In this chapter we presented Concerto, a framework for soundly combining concrete and abstract interpretation. Concerto targets framework-based applications that use difficult-to-analyze reflection, metaprogramming, and abstractions. This combination is possible because framework-based applications in practice satisfy a state separation hypothesis which Concerto exploits to opaquely embed abstract values into a mostly-concrete interpreter. Our combination supports any abstract interpreter that satisfies a modest set of conditions, and yields significant improvements in initial experiments with a research prototype.

Chapter 7
SUPPORTING THE FULL JVM

7.1 Introduction

The combined interpretation approach of Concerto described in Chapter 6 is a first step in supporting the analysis of framework-based applications. As mentioned in Section 6.9, the Yawn framework used in Concerto's evaluation supported many features found in real-world Java frameworks. However, the language supported by Concerto is significantly simpler than Java. The following is a non-exhaustive list of Java features Concerto does *not* support:

1. Strings
2. Exceptions
3. Concurrency
4. File I/O
5. Native Extensions
6. Subclassing
7. Class Loaders
8. *invokedynamic*
9. Full Reflection API

Although Yawn used a simple reflective facility provided by our proof-of-concept implementation, this API had considerably less functionality than Java's. Similarly, the input primitives of the Concerto implementation provided read access to only two

streams of integers, one of which was deterministic, the other nondeterministic. The only output primitive wrote objects to the console; we did not support reading or writing files. These facilities are considerably simpler than the I/O features provided by Java, which support opening network sockets, reading and writing arbitrary file systems, etc.

Extending the existing Concerto implementation to handle the full complexity of Java, while technically possible, is likely infeasible in practice. The Concerto interpreter is written in Java and uses the `HashMap` class to represent a program's heap. As a result, the host JVM memory overhead for each object allocation in a guest program is extremely high. The lattice and widening operations on heaps also walk the entire concrete object graph. Although Yawn and our simple web application were not large or complex enough to manifest any significant slowdowns, it is unlikely the simple approach taken by Concerto would scale to a large, full-featured Java framework.

Further, even if we extend Concerto to support the full set of Java's bytecode instructions, the resulting system would be unable to run many Java applications. Almost every Java application relies on at least some parts of the Java Class Library [1], which internally uses several native operations that must be supported by the host JVM implementation. Thus, using existing Java Class Library distributions (such as the one shipped with OpenJDK) on Concerto would require several hundred native method implementations.

Due to the drawbacks sketched above, we have begun preliminary work on a project to support the full JVM and Java language that is not based on our Concerto implementation. This project (called Symphony) builds on the infrastructure developed for Java Path Finder (JPF) [30]. Unfortunately, supporting the full Java language, Java Class Library, and JVM features is a multi-year effort that is likely worthy of its own dissertation. This chapter describes the current state of Symphony and the work remaining. Section 7.2 describes the features of Java Path Finder that make it a suitable starting point for the Symphony project. Section 7.3 describes the state of the Symphony project and the technical achievements accomplished thus far. Section 7.4 describes a novel analy-

sis scheduling algorithm we have developed for use in Symphony. Finally, Section 7.5 describes the technical, engineering, and research challenges that must be overcome to complete the Symphony project and sketches potential solutions.

7.2 *Java Path Finder*

Java Path Finder was developed as an explicit state model checker for Java programs, although later work generalized the infrastructure to support symbolic values. Like Concerto, Java Path Finder is an implementation of the JVM written in Java that itself runs on a host JVM. Java Path Finder supports several features necessary for combined interpretation over the full Java language that are currently missing from or naïvely implemented in Concerto:

1. **Heap Reification**: Unlike techniques built on the JVM (see below), Java Path Finder has a well tested, efficient in-memory representation of concrete heaps and objects that support freezing and copy-on-write. When embedding a concrete heap into an abstract interpreter's state (see Section 6.8.1) we can simply embed a reference to an immutable instance of JPF's heap representation.

2. **Execution Rollback**: JPF can save the guest JVM state at arbitrary points into a self-contained *memento* object. JPF can use a previously saved memento to restore JVM state to the exact machine state when the memento was created. Unlike native approaches (discussed below), these mementos are first class objects and can be embedded into an abstract interpreter as a continuation or into a worklist for processing.

3. **Execution Forking**: When analyzing a program, JPF uses *choice generators* to exhaustively explore multiple possible paths of execution. Each choice generator represents a finite number of possible choices a program could make, e.g., possible program input values, thread scheduling choices, etc. The Symphony project

can leverage this choice generator infrastructure to support forking execution at nondeterministic conditionals as described in Section 6.8.3.

4. **Symbolic Values**: JPF supports an attribute API, which tracks arbitrary data attached to program values, threads, stack frames, and objects. JPF automatically transfers and duplicates these attributes as host values move through the program. We can use this infrastructure to embed abstract values or sets of concrete values into the concrete state.

5. **Native Library Support**: The JPF project includes implementations of a large portion of the native methods depended upon by the Java Class Library. These native method implementations are written in Java, run in JPF's host VM, and have access to the internal state of the simulated VM.

6. **Extensible Semantics**: JPF allows replacing the standard semantics of bytecode instructions with instrumented versions. This functionality allows us to naturally extend operations over concrete values to handle abstract values, intercept method calls with abstract receiver objects to switch to abstract interpretation, etc.

We briefly considered adapting techniques that do not require a reimplementation of the JVM. In particular, we briefly experimented with the CROCHET tool [18], which supports snapshot and rollback on stock JVMs. Under an approach based on CROCHET, framework code could execute on a native JVM, using CROCHET for forking and rollback support. Running framework code on the native JVM would not require reimplementations of the native methods used by the Java Class Library. Further, CROCHET has a very low overhead which could yield better performance over running the framework code inside of a simulated JVM that itself runs on a JVM.

However, despite the above benefits, we ultimately rejected this approach as CROCHET does not (at the time of writing) support first class snapshots: the saved program

state is stored in hidden object fields inserted via bytecode instrumentation. CROCHET also cannot support multiple snapshots out-of-the-box, nor do we gain the benefit of first-class heap snapshots.

Beyond the apparent limitations of CROCHET, running the framework code on a standard JVM itself has several drawbacks. First, supporting symbolic values (i.e., embedded abstract values and the mostly-concrete representation of framework values) would require heavyweight bytecode instrumentation. In particular, storing symbolic primitive values would require the expensive (in terms of storage and runtime overhead) Phosphor [17] instrumentation used in Staccato. Instrumenting the framework implementation to correctly manipulate and handle these symbolic values would require even further instrumentation. It is likely that the overhead introduced by these multiple instrumentation passes would erase the performance benefits of running framework code on a "native" JVM. Finally, based on our experience with Staccato, the composition of multiple low-level bytecode instrumentations is extremely tricky and requires close cooperation between all instrumentation passes. Thus, we rejected all techniques involving running the framework code directly on a stock JVM.

7.3 *Current Progress*

Symphony supports a significant subset of the mostly-concrete semantics implemented within Concerto. In particular, our implementation supports: sets of potential values (i.e., $\mathbb{P}(V_f)$), the special unknown $\top$ value, summary objects, loop approximation, and forking/joining paths of execution. In the remainder of this section, we briefly sketch how we have implemented these features within the JPF infrastructure.

7.3.1 *Value Sets and Heaps*

Symphony uses the attribute API to record when fields/variables/etc. have multiple possible values or a completely nondeterministic value. Symphony overrides the default se-

mantics for bytecode operations to handle nondeterministic operands or operands with multiple possible values. As in Section 6.8, we lift the arithmetic operations to handle all combinations of potential input values. Further, any arithmetic/comparison operation with at least one nondeterministic operand yields a nondeterministic result.

For load operations with multiple possible base pointers, SYMPHONY combines the results of reading the field from each possible base pointer. During a field store, multiple base pointers represent uncertainty about the precise object whose field is being updated. Thus, SYMPHONY *weakly* updates the field of each possible object. To handle array operations, SYMPHONY must also account for nondeterministic or multiple possible array indices. In the case of array loads, SYMPHONY collects the values at all possible array indices (or all array indices for a nondeterministic index operand) from all possible array pointers. For array stores, SYMPHONY performs a weak update of all possible indices for all possible array pointers. Updates on summary objects (introduced by widening, see below) are always weak. Note that these nonstandard semantics are straightforward to implement due to JPF's extensible bytecode semantics.

To represent the concrete heap, we adapt the approach used in CONCERTO. For each object allocation site, we compute the allocation context which consists of the stack trace from `main()` to the allocation site and the allocated type. We associate each context with a counter. At each allocation in a context C, we increment C's counter yielding the integer v. The tuple $\langle C, v \rangle$ serves to uniquely identify an allocation, i.e., a heap location. The default heap implementation provided by JPF uses a similar approach, so adapting CONCERTO's approach to the JPF infrastructure was relatively straightforward. We further extend the above scheme to handle object allocations within nondeterministic loops. Our widening operator detects if an allocation context C occurs within a nondeterministic loop and sets the corresponding counter to $\top$. Any future allocations at C will return the heap location $\langle C, \top \rangle$ which points to a summary object. This summary object represents the infinite set of objects allocated at C. As such, all updates on a summary object are weak.

7.3.2 Forking/Joining Execution

As mentioned in Section 7.2, JPF has extensive support for forking a single path of execution to consider multiple potential runtime behaviors. Symphony uses this infrastructure to fork execution at nondeterministic conditionals and to handle method calls where the base pointer has multiple possible values. However, *joining* paths of execution required significant extensions to the JPF implementation.

Java Path Finder forks execution upon registration of a *choice generator*, an object that represents different possibilities a program could encounter during execution, e.g., different possible thread schedulings, user input values, etc. The choice generator author must also instrument the program semantics to interpret these choices and guide the program down the selected path, e.g., schedule the chosen thread for execution or push the potential input value on the stack.To handle nondeterministic conditionals, Symphony registers a choice generator which simply enumerates the possible conditional values true and false, and instruments the conditional semantics to select either the goto target or fallthrough successor based on the chosen value. To handle method calls with multiple possible receivers, Symphony registers a choice generator which enumerates over all possible receivers; the instrumented method call semantics used by Symphony then calls the resolved method on the chosen receiver object.

Although we can use choice generators to fork execution, joining two paths of execution is not possible in stock JPF. This deficiency does not impact soundness, only termination. Without joining, JPF can quickly encounter the exponential path explosion common in symbolic executors or diverge in the presence of unbounded loops. We have therefore extended JPF to merge forked paths of execution that reach common program points, which we call *join points*. In the presence of `continue`, `break`, and `return` statements, finding these join points is nontrivial; we provide a complete overview of the algorithm in Section 7.4. At a high level, when execution reaches a join point, Symphony creates a snapshot and pauses execution. When another path of execution reaches the same join

point, JPF joins that path's state with the currently waiting snapshot. This join operation is itself defined in terms of join operations we have implemented over snapshots of the heap, thread state, etc. After joining, Symphony continues execution from this newly created snapshot, using the existing JPF resume execution functionality. Finally, to handle a method call with multiple possible receivers, we join the paths spawned for each callee at the control-flow successor of the method call.[1]

7.3.3 *Loop Approximation*

As explained in Chapter 6 (in particular, Sections 6.7 and 6.8.3), framework implementations may contain statically unbounded loops, and without an ACC property for the semantic domain, the mostly-concrete interpreter may fail to terminate when executing such loops. Thus, Symphony detects potentially statically unbounded loops and applies a *widening* operation. This widening operation is an almost direct translation from the one used in Concerto, modified to operate over JPF's snapshot representation. Symphony widens snapshots that reach the heads of an unbounded loops.[2] When we detect that the state at the loop header has stabilized across iterations, we prune any further executions of the loop that return the header, continuing execution only on paths that exit the loop.

7.4 *Scheduling Execution*

As mentioned above, joining multiple paths spawned at nondeterministic conditionals is a surprisingly subtle problem. In a high-level language with source-level conditional and loop expressions, the problem is (relatively) straightforward. However, JPF operates over

[1] In principle, we could have used a similar strategy to handle heap read/writes with multiple possible base objects. However, each choice generator creates a snapshot of the entire execution state which is relatively expensive to construct.

[2] The test for unboundedness is *dynamic*, in that we first assume every execution of a loop is bounded until we detect that the loop may cause our interpreter to diverge. Thus, an execution of a loop may be judged to be unbounded in one state, and bounded in another.

```
lbl: {
  if(⋆) {
    // ...
    if(e) {
      break lbl;
    }
    // ...
  } else {
    // ...
  }
  print("hello"); // point a
}
print("world"); // point b
```

Figure 7.1: Contrived example illustrating multiple join points. The two threads of execution spawned at line 2 may join at point a or point b depending on whether `e` evaluates to true or false.

Java bytecode which only supports conditional/unconditional gotos for control-flow, significantly complicating the task. This section provides a formal problem definition, our proposed solution, and a sketches a proof of correctness for our technique.

7.4.1 *Motivation and Problem Definition*

When JPF reaches a nondeterministic conditional, it forks execution and continues execution down both the true and false branches. This exploration does *not* happen in parallel: JPF uses a *search strategy* to select which path of execution it will explore. For example, the depth-first search strategy included with JPF will fully explore all execution paths starting from the true branch, backtrack, and then fully explore paths spawned from the false branch. Unfortunately, neither this strategy nor any other implemented in JPF can join paths of execution. In the simple case, a merging strategy must execute one branch until it reaches a join point j, pause execution, backtrack, and then execute the other branch until it also reaches the join point j. Once both executions have reached the same join point, JPF may join their states and resume execution.

However, as will soon be apparent, it is not statically apparent where exactly the join point for a nondeterministic conditional will be. Consider the contrived example shown in Figure 7.1. The conditional on line 2 is nondeterministic (as indicated by the $\star$). The conditional line 4 *is* deterministic, and its true branch contains a labeled break statement which transfers control flow out of the block labeled `lbl`. Thus, the two paths of execution may meet at one of two points: point a if the conditional `e` on line 4 evaluates to false, or point b if it evaluates to true. Thus, we must consider *sets* of potential join points when merging execution.

Simply computing sets of join points is necessary but not sufficient. Let us return to our example in Figure 7.1. Suppose after forking execution JPF chooses to explore the false branch; after executing the else block it reaches point a, which has been identified as a join point. As described above, JPF will suspend execution, backtrack and begin ex-

ecuting the true branch. Suppose now that when execution reaches line 4 the expression `e` evaluates to true and control-flow exits the labeled block, reaching point b. As point b is a control-flow join point, JPF will again pause execution. However, both sides of the conditional have been explored, so both paths of execution are now stuck! As an execution starting from a can reach b (but not vice versa) we must resume execution at a, eventually joining with b. As this example illustrates, our algorithm must also calculate dependency information between join points. Our approach uses this dependency information to resume an execution waiting at point j when there are no other outstanding executions that may reach j.

This section describes algorithms for computing the control-flow join points, the dependency information between them, and how to consume this information when scheduling execution. In the following, we will say a *thread* of execution to refer to some JPF VM state with a program counter, concrete heap, stack, etc. A thread may be suspended (i.e., held in a worklist) or running (i.e., actively executing on the simulated JPF VM). We will refer to a *a thread waiting at point* p to describe a suspended thread of execution whose program counter is p. We say a *path of execution* π to describe a concrete path taken by some thread of execution through a method's control-flow graph. Due to forking, multiple paths may originate from a thread t. Finally, we will often use the name of a program point to refer to a thread of execution paused at that point ("e.g., join with a").

Before continuing, we outline two more properties that our algorithm should satisfy. Our scheduling should be *loop consistent*, which intuitively requires that threads on different iterations of the same loop should never be joined. In other words, a thread t executing the ith iteration of a loop should never be joined with another thread t' executing the jth iteration of the same loop (where $i \neq j$). In fact, our algorithm guarantees that all active executions within a loop must be on the same iteration. This property is not a correctness condition, but ensures loops are handled uniformly. Formally, we associate with each thread of execution a vector of loop iteration counters $\mathfrak{I}$. Each position

```
i=0;
while(...) {
  if(i == 1 || ⋆) {
    // (a)
    return ...;
  } else {
    // (b)
    i = 1;
  }
}
```

Figure 7.2: Example of loop consistency ruling out a join point. Joining the thread of execution at b with a would require beginning another iteration of the loop, which violates loop consistency.

in the vector corresponds to a loop in the executed method. A value of ι at a position in $\mathfrak{I}$ corresponding to a loop l indicates that the thread is not executing l. A value of $i \in \mathbb{N}$ indicates that the thread is on the ith iteration. Reaching the loop header sets the counter to 0; entering a loop either increments the loop counter, and exiting the loop resets the counter ι (a fully formal definition is found in Appendix C). In a loop consistent execution state, every thread's loop vector has ι or the same value i at each loop position. Requiring loop consistency may rule out some join opportunities. Consider the code fragment in Figure 7.2. Suppose there are two threads of execution paused at points a and b respectively; each thread is on the ith iteration Further, suppose that b re-enters the loop when executed. If we resume the thread at b, after re-entering the loop it will deterministically take the true branch and join with a. However, allowing b to go around the loop and join with a would violate loop consistency. In fact, in the given example, there is no point at which the threads a and b may rejoin without violating loop consistency.

```
if(*) {
  // ...
} else {
  // ...
}
int x = expensive(); // point a
int y = 5; // point b
```

Figure 7.3: Example of multiple join points. It the paths spawned at the non-deterministic conditional are joined at point a, then `expensive()` will be executed only once. If the paths are joined at point b, `expensive()` will be executed twice, increasing analysis time.

Second, as a performance consideration, we would like our control-flow joins to be *early optimal*. Intuitively, an early optimal join is when two paths of execution are joined at the earliest possible point. For example, consider the code in Figure 7.3. After forking at the conditionals, the earliest the two paths of execution merge is at point a. It is technically correct to identify point b as the merge point, but doing so will cause `expensive()` to be executed twice, potentially increasing analysis time.

7.4.2 *Preliminaries*

We now describe how we compute information used by the rest of our approach. For the remainder of this section, we will assume an exception free control-flow graph for a specific method, with successor and predecessor relations *succ* and *pred* respectively. We do *not* need to join paths across method boundaries; all threads are joined at method return before control is transferred back to the caller. In the following, we will refer to abstract "points" in the control-flow graph, leaving ambiguous whether we are referring to join basic blocks or bytecode instructions. It turns out not to matter which we mean,

although our algorithm is ultimately implemented in terms of basic blocks.

Identifying Loops To find the loops in a method we use the standard natural loop detection algorithm [4]. We compute the dominators in the control flow graph. If a basic block b has a successor s such that s dominates b, b is in a loop with s as a header. We will write $\mathsf{doms}(s, b)$ to denote that s dominates b. We say that the CFG edge from b to s is a *back jump*. We compute the loop body by traversing the predecessor edges of the CFG from b until we reach the header s. For simplicity, we assume a loop has a single entry point. We could relax this restriction by using the technique described in [139]. We further assume that any cycles in the control-flow graph are the result of well-nested, natural loops.

Our definitions will use the following formal definitions. In the following $\mathcal{R}^+$ denotes the (non-reflexive) transitive closure of relation $\mathcal{R}$ and $\mathcal{R}^*$ is the reflexive, transitive closure:

$$\mathcal{L}(h) = \{p \mid p \in \mathsf{pred}^*_{-h}(n) \land \mathsf{succ}(n, h) \land \mathsf{doms}(h, n)\} \tag{7.1}$$

$$\mathcal{H}(b) = \{p \mid b \in \mathcal{L}(p)\} \tag{7.2}$$

$$\mathsf{pred}_{-h} = \mathsf{pred} \setminus \{(q, h)\} \tag{7.3}$$

$\mathcal{L}(h)$ is the set of statements/blocks in the body of the loop with header h, and $\mathcal{H}(b)$ is the set of headers of loops that contain b. Note that $\mathcal{L}(b) = \emptyset$ if b is not a loop header.

Finding Join Points We do not maintain a per-conditional list of join points, but rather a set of all join points in the entire method. Identifying join points is relatively straightforward: we simply find all points with 2 or more predecessors. Note that this definition necessarily includes all loop headers in a method.

7.4.3 *Identifying Dependencies*

After we have identified the join points of the method, we compute the dependencies between them. We say a join point p *depends on* q iff a path of execution at q may reach p without violating loop consistency. Recall that our goal is to avoid executing the same code within different threads of execution. If we scheduled p while other threads could still reach p then we would potentially re-execute the same code. Thus, we will execute a thread paused at p only if there are no threads paused at p's dependencies.

We compute the *dependents* of a join point as follows. Define acyclic successors as:

$$\mathit{succ}_{\downarrow} = \{(n, m) \mid (n, m) \in \mathsf{succ} \wedge \neg \mathsf{doms}(m, n)\}$$

For any point p, $\mathit{succ}_{\downarrow}^{+}(p)$ is the set of points reachable from p without following a backjump, i.e., without looping. Notice that $\mathit{succ}_{\downarrow}$ is acyclic (if it was not, we would have a loop in the control-flow graph not involved in a natural loop, violating our assumption). Define the immediate dependents of p as:

$$\mathfrak{D}(p) = \left(\mathit{succ}_{\downarrow}^{+}(p) \cup \mathcal{H}(p)\right) \setminus \mathcal{L}(p)$$

The full dependents are $\mathfrak{D}^{+}(p)$. The immediate dependents of a point are those acyclically reachable from p (excluding the loop body of p, if any) and the loop headers of the containing loops of p. Intuitively, if p can reach q acyclically, then q should wait for p (the $\mathit{succ}_{\downarrow}^{+}$ term). Further, executions waiting at the head of a loop must wait for any executions within the body of loop; that is, any completed iterations of a loop must wait for other in-progress iterations (the $\mathcal{H}(p)$ term). Note that $\mathfrak{D}$ is acyclic. To see why, consider that the base relation $\mathit{succ}_{\downarrow}$ is acyclic. The only additional edge added is from a point b within a loop to its headers ($\mathcal{H}(b)$). However, we remove any edges from any loop headers to their bodies (the subtraction of $\mathcal{L}(p)$) ensuring no cycle is created.

7.4.4 *Scheduling Executions*

Given dependent information, scheduling executions is relatively straightforward. At each scheduling point, JPF selects a thread of execution t paused at a join point p with no active dependencies: i.e., there are no threads of execution paused at a point q such that $p \in \mathfrak{D}^+(q)$. As $\mathfrak{D}$ is acyclic, a least one paused thread must satisfy this requirement. JPF then resumes execution from t pausing when t (or any threads forked from t) reach *any* join point. After all threads have paused the above process is repeated, selecting an unblocked thread and resuming execution. This process repeats until all threads have returned from the method.[3]

7.4.5 *Correctness*

We have an preliminary proof of correctness for the above approach. We have proved that $\mathfrak{D}^+(p)$ contains all of the join points reachable from p that are eligible as loop consistent join points. In other words, the points excluded from $\mathfrak{D}^+(p)$ cannot be loop consistent, early optimal join points for p. We have also proved that any scheduling admitted by the above approach will be loop consistent. These proofs are found in Appendix C.

7.5 *Future Challenges and Potential Solutions*

As described in the previous two sections, significant progress has been made supporting mostly-concrete semantics in Symphony. We now describe the work remaining to achieve feature parity with Concerto (Section 7.5.1), and the expected challenges we expect to face scaling Symphony to the entire JVM (Section 7.5.2).

[3] As mentioned in Section 7.3.3, some loops may not terminate without widening. In practice our scheduling algorithm integrates with the machinery described in Section 7.3.3 to widen states at the headers of divergent loops.

7.5.1 Concerto *Feature Parity*

There are three key features of Concerto yet to be implemented in Symphony: yielding into an abstract interpreter, yielding from an abstract interpreter into a concrete interpreter, and embedding abstract values into the mostly-concrete interpreter. The final feature, embedding abstract values, is relatively simple to implement using the same attribute API used to support multi-values and nondeterminism described in Section 7.3.1. Similarly, we can embed the abstract heap into the mostly-concrete state by attaching a reference to the abstract heap as an attribute to a dummy object allocated in the concrete heap.[4] We therefore focus on how we plan to support the first two features: yielding from the abstract interpreter into the concrete interpreter, and vice versa.

We first describe how we plan to implement calls from the abstract interpreter. Our instrumented method invocation semantics can detect when the receiver is an embedded abstract value using the attribute value API discussed above. As in Concerto, Symphony will invoke a method exposed by the abstract interpreter, passing in the abstract receiver, method signature, method arguments (with any embedded abstract arguments unwrapped), and information about the caller execution state. Symphony will also pass in the abstract heap (extracted from the dummy object described above), a concrete heap snapshot, and a snapshot of JPF execution state (including stack and thread snapshots) for resuming concrete execution when the abstract interpreter finishes analyzing the callee.

Yielding from the abstract interpreter into the concrete interpreter is also greatly simplified by existing JPF infrastructure. JPF supports "direct call frames" which represent call frames pushed by a method call in "native"[5] code. When the abstract interpreter encounters a method call with a concrete receiver, it will call a method provided by Symphony, passing in the method name, receiver, arguments, caller state, etc. Symphony

[4]We assume, as in Concerto, that the abstract heap implementations are copy-on-write or immutable. We can relax this assumption by requiring abstract heaps support a "clone" operation.

[5]Recall that native method implementations in JPF are written in Java.

will extract the abstract and concrete heaps, construct a fresh thread state with an empty stack, push a direct call frame onto the stack, and then begin concrete execution. After the concrete execution returns, SYMPHONY will dispose of the thread state, and return the updated concrete/abstract heap and return value to the abstract interpreter.

7.5.2 *Challenges and Proposed Solutions*

We now describe the key challenges we expect to encounter during the implementation of SYMPHONY. We expect that some of the missing features described in Section 7.1, such as strings, subclassing, and exceptions, will be relatively straightforward to implement. For example, we can support exceptions by extending the API used to communicate return values between interpreters to include a list of exception type/object reference pairs for each exception that may escape a called method.[6] We therefore focus on what we view as the most challenging questions:

1. How do we scalably and concisely represent method summaries?
2. How do we use existing native method implementations that manipulate a mostly-concrete/abstract heap?

We now elaborate on these questions, and our proposed solutions.

Scalable Method Summaries

Caching method summaries is crucial to static analysis performance [7]. One of the simplest caching strategies used in most abstract interpreters is simple input/output caching. Input/output caching records the abstract state s' for a method m analyzed in an initial state s under context C. If the analysis ever encounters a call to m under context C with initial state $s'' \sqsubseteq s$, then analysis of m can be skipped and the previously

[6] We expect this approach will use the combined object reference representation used for interface types described in Section 6.8.4.

cached result of s' used instead. However, if the initial state s contains a concrete heap component (as is the case with Symphony and Concerto), then even a single bit change anywhere in the heap will trigger a potentially expensive reanalysis of a method. Further, even checking that an initial state is subsumed by a previously encountered input state requires walking the entire concrete heap, itself a very expensive operation.

We propose a solution based on *read field* sets, which records for each method m the set of object/field pairs read by the method. When checking for input state subsumption, only the object/field pairs read by the method need to be checked. We expect that the majority of methods only access a small portion of the object graph (namely the objects transitively reachable from the input parameters), therefore in the common case checking for input state subsumption will be relatively efficient compared to the naïve approach.

However, a simple input/output caching strategy with read fields will be unsound. To see why, consider a method m that reads a single object field f. Suppose this method has been invoked in a heap where $f = 1$ and the value of some field unrelated to the method is 2. As g is untouched by m, our cached output state contains a heap where $g = 2$. Finally, suppose we encounter a call to m in a state where $f = 1$ but $g = 3$. Our naïve read set based strategy would compare the values of f, and then use the cached method result. However, this cached result has the incorrect value for g: the 3 in the caller state has been mysteriously replaced with 2.

We therefore extend the above technique with *write sets*. During method execution, we can instrument JPF to record all values *written* to object fields. This information can be cached with the input/output summary. If the read field values in some caller's state are subsumed by a previous execution, then the caller's heap can be updated in place with the cached write information. These summaries must also account for object allocations that occur during execution, but these can be overapproximated using the summary objects described in Section 7.3.1.

An even more efficient form of method summaries are *symbolic summaries*, which compactly represent the behavior of a method as a function over input data. Symbolic

summaries can yield significant speedup but are significantly more challenging to implement compared to simple input/output caching for heap-manipulating methods. One simple (albeit imprecise) symbolic summary based on the write sets described above is to simply havoc all object fields written during method execution. This summary may be used instead of executing potentially expensive methods in the framework. For example, Symphony could collect run time statistics for framework methods and use this imprecise summary for methods with relatively small write sets but long execution times. An additional benefit of such imprecise summaries is that their representation is relatively simple and compact, and can be soundly and concisely reused within a symbolic summary of application code that calls into framework code.

Native Method Stubs

JPF includes implementations for the native methods relied upon by the JCL. These implementations are written in Java and, like their native VM counterparts, directly mutate object fields, thread state, and other VM internals. Unfortunately, these implementations only handle fully concrete heaps/objects/etc. For example, the JPF implementation of `System.arraycopy` only copies the concrete array contents from the input array. Using this implementation in conjunction with abstract values will be unsound, as it fails to copy any symbolic values attached as attributes to the source array. Similarly, the implementation of the reflective `Field.set` operation unconditionally *strongly* updates an object field, which would be unsound if the referenced object is a summary object.

We now sketch possible solutions for handling native methods that read or write values in the heap. These solutions have different soundness/precision trade-offs. The simplest is to manually extend the APIs used by the native method implementations to record the object/fields updated during native method execution. After the native method finishes, Symphony could (pessimistically) havoc these updated fields. This approach is relatively simple to implement, while retaining soundness at the cost of

some precision. At the other end of the soundness/precision spectrum, we could on-the-fly optimistically generate a concrete value *consistent* with any symbolic values read by the native method stub. For example, if a native method reads an object field with three possible values, SYMPHONY could instrument the field read API used by the native method to nondeterministically select one of the values. This approach is also very simple to implement, but could undermine the soundness of SYMPHONY, depending on the frequency that native methods access symbolic values. Finally, as the native method implementations are written in Java (and thus compiled to Java bytecode), they are amenable to bytecode instrumentation that adds shadow state which transparently tracks any symbolic information (i.e., embedded abstract values, multi values) about program values accessed by the native method implementations. This approach can preserve soundness and precision at the cost of significant implementation complexity and performance overhead. Recent techniques developed by CROCHET have shown how to add *and remove* instrumentation during program execution which can mitigate any performance impacts.

The above solutions only handle native methods that *unconditionally* read or write values. A native method may make control-flow decisions based on values read from the heap or parameters. For example, consider again the `Field.set` method. The specification requires that the implementation throw an exception if the first argument, i.e., the object whose field value should be updated, is `null`. Further, multi-value references are attached to the sentinel dummy value `null`. Thus, if `Field.set` is called with a multi-value argument, the native method implementation will observe the dummy `null` value of the argument leading to an incorrectly thrown exception.

One potentially precise *and* sound approach that addresses the above problem is to concretely execute the native method with all possible valuations of multi-value arguments and join the results. Returning to our `Field.set` example, suppose we reach a call with arguments $\langle\{\ell_1, \ell_2\}, \{1, 2\}\rangle$, where ℓ_1 and ℓ_2 are concrete addresses. We could then concretely execute the field update with all the arguments in $\{\langle l, i\rangle | l \in \{\ell_1, \ell_2\} \wedge i \in \{1, 2\}\}$.

In the simplest case, this approach requires $\Pi_i|v_i|$ executions for an invocation of a native method with arguments $\langle v_1, \ldots, v_n \rangle$, where $|v_i|$ is the number of possible values for argument v_i. There are two major drawbacks with this approach, First, a native method may read the value fields with multiple possible values during execution. It is impossible to fork execution within the native method implementation. Instead, Symphony must abort execution of the native procedure and fork execution before the native call, cycling through all possible values of the read field. Second, for many abstract values it is be impossible enumerate all represented concrete values. Symphony may have to abort when an abstract value is read and fall back on less precise approaches described above. This approach is relatively simple to implement with precision and soundness benefits for certain calling patterns. However, it could potentially require executing a native method implementation multiple times for each source program invocation and may not be universally applicable for all native method invocations.

7.6 Conclusion

The Concerto project demonstrated that abstract interpretation can be applied to applications written against difficult-to-analyze framework code. For the results of Concerto to be practically useful, significant work must be done to support the full complexity of the Java language and JVM. This chapter outlined the progress we have made on the Symphony project, which has reimplemented significant portions of the mostly-concrete semantics of Concerto within JPF. It also contained a description of a novel analysis scheduling strategy used within Symphony. We sketched our implementation strategy for the remaining Concerto features needed by Symphony, and identified two key research challenges that must be solved to bring combined interpretation to the full Java language: scalable method summaries and native method execution with symbolic values. We have sketched multiple possible solutions to these challenges, although the ultimate solutions will depend on the prevalence and impact of these challenges.

Chapter 8

RELATED WORK IN WHOLE-PROGRAM STATIC ANALYSIS

Frameworks As described in Chapter 5, other researchers have tackled the challenge of analyzing framework based applications. In particular, the popular Android mobile application framework has received particular attention given the ubiquity of the platform. Many techniques use model (sometimes called "harness") generation, i.e., the generation of dummy main methods that summarize the behavior of the framework [8, 132, 79]. The model generation approach has been applied to other frameworks. For example, Livshits and Lam analyzed the securibench suite [125] by generating "invocation stubs" [129]. LEGATO also uses model generation as described in Chapter 5. In principle, harness generation simplifies analysis of framework-based applications by creating an easy-to-analyze program which can be understood by any off-the-shelf analysis. In practice, a sound harness alone is insufficient to precisely analyze framework-based applications. As described in Section 5.2.3, frameworks provide functionality beyond a main event or dispatch loop, such as session and state management, indirect flow, etc. Precisely handling these features requires manual modeling or complete and precise framework specifications. In fact, as noted in [23], a large part of the FlowDroid project was the development of the *taint wrappers* which summarize the flow of information through frameworks and libraries. Unfortunately, the effort of creating a model generator and precise method summaries is *not* a one-time cost: changes to framework adding or changing functionality require corresponding changes to model generators and summaries.

In place of harness generation, other analyses have opted to embed models of framework behavior into the analysis itself [190]. Another approach developed in the Droidel

project [23] avoided models entirely, instead focusing on *explicating* framework behavior. Specifically, the authors *manually* replaced instances of difficult-to-analyze features (e.g. reflective allocations, native code methods) in the framework with invocations to special Droidel stub methods. Based on an application's configuration file, the Droidel tool generates implementations for these stubs that soundly summarize the replaced feature for the application. For example, reflective allocations of an Android `Activity` object are replaced with calls to a generated stub method that instantiates `Activity` objects registered in the application's configuration file. Unfortunately, this proposed approach still requires manual changes whenever the framework is changed. Further, as Droidel explicitly focuses on explicating *only* Android's use of reflection and native code ([23, Section 3.1]) it does not enable the precise analysis of the framework features cited above. For example, Droidel models one of Android's implicit flow features by generating a large switch statement. Precisely resolving indirect flows using this model would thus require context-sensitive constant propagation with static resolution of switch statements. In other words, the client analysis must compensate for an imprecise model with significant analysis machinery.[1]

The Frameworks for Frameworks (F4F) system by Sridharan et al. [174] generalizes these efforts, providing a framework for writing framework models in a DSL called WAFL. These models specify, e.g., how to replace calls to framework APIs with simpler, equivalent code. In practice, these DSL models are generated based off an application's configuration file as in harness generation. Thus, using F4F to model a new framework ultimately requires producing a generator after becoming an expert in the framework functionality. Further, it is unlikely that the model generators created for one framework can be meaningfully reused within another framework. Although F4F can *reduce* the time to develop a framework model generator (to reportedly as low as 1 week per framework), modeling frameworks remains a non-trivial time investment that must be paid to support

[1] The authors note that Droidel can complement existing modeling techniques, which implies Droidel is a *partial* solution for analysis of framework-based applications.

any new frameworks.

Although the above techniques have different trade-offs, they all fundamentally rely on some form of manual modeling and framework expertise. In contrast, once the concrete semantics for a language are specified, combined interpretation can in principle be used with any framework without further effort. In practice, both Symphony and Concerto rely on oracles to determine: 1. which I/O operations are nondeterministic, and 2. which classes belong to the framework vs. application. In the former case, we can develop a single oracle for all I/O primitives included in the Java Class Library. This oracle could be configured to consider specific files/folders/hostnames as deterministic sources, treating all other sources as nondeterministic. Although providing this configuration is some manual effort, it is significantly less work than creating an exhaustive model of framework behavior. Similarly, we can develop a single oracle which is configured with the sets of framework and application jars; classes loaded from an application jar are considered application types, and vice-versa for classes loaded from framework jars. This configuration is a relatively small piece of manual effort, and we expect that developers can easily identify which JAR files comprise the framework vs. the application.

Reflection Concerto and Symphony aim to precisely analyze framework implementations, which often entails precisely resolving reflective operations. Several static analyses have been developed to exclusively resolve reflection. For example, [119, 171] collect type constraints on reflective operations to narrow the space of possible callees/instantiated objects. For example, if the result of a reflective instantiation is downcast to type I, then these analyses constrain the reflective instantiation to only construct classes that subtype I. Both analyses cited above also track constant strings that flow to the Java reflective API to further refine the resolution of reflective calls. Work by Barros et al. [15] have used the Checker framework [64, 150] to exploit static string information to resolve reflective invocations. However, all of these techniques rely on reflection APIs being used

in stylized ways. These patterns, particularly the use of constant strings, rarely occur in framework implementations. In addition, these techniques are tailored to specific APIs (i.e., Java's reflection API) and adapting them to other domains requires non-trivial work by the analysis designer.

Combined Analyses Many efforts have combined dynamic and static analysis, yielding "blended" or "hybrid" analyses. Some hybrid analyses use information recorded during dynamic executions of the program to improve the static-analysis precision [88, 68, 199, 56, 162]. For example, TamiFlex by Bodden et al. [26] instruments an application to record the callees of reflective calls as it runs a representative workload. These approaches generally suffer from unsoundness; even representative workloads rarely exercise all possible execution paths. Symmetrically, other researchers have run a dynamic analysis seeded with information produced by a static analysis [14, 113, 44]. As with the above approaches, any dynamic analysis will almost certainly be unsound.

Other researchers have explored other approaches to combining analyses. For example, [74] runs a series of increasingly precise analyses to prune false positives left by earlier analyses. Work on tunable analyses of JavaScript [108] uses the results of a pre-analysis to restrict the abstract states explored by an abstract interpretation. Ferrara et al. [73] explored combining a value and heap analysis to improve precision in abstract interpretation. Their approach embeds information produced by the heap analysis into the domain of the value analysis. This embedding closely mirrors how Concerto embeds abstract values into the mostly-concrete state and vice versa.

Other researchers have combined different execution strategies for different portions of code [12, 43]. For example, Chipounov et al. [43] switch to *fully* concrete execution on portions of the system under test. Their approach requires concretizing symbolic values and then checking that no feasible paths are pruned as a result. Concretization is performed lazily, which parallels how Concerto threads abstract values through mostly-concrete interpretation. However, due to the state separation hypothesis, Concerto can

avoid almost *all* concretization. Godefroid et al. [87] have developed the SMASH algorithm which exploits an insight similar to that behind Concerto to effectively interleave may- and must-analyses. Specifically, they note must-analyses (i.e., directed concrete execution) can easily reason about code that would confound a may analysis, and similarly may analyses can easily prove properties about code that confounds must-analysis. Their verification technique composes may and must summaries to gain precision and performance improvements over using a single technique. This interleaving of concrete and symbolic reasoning has clear parallels to our interleaving of mostly-concrete and abstract interpretation on different portions of a system.

Finally, many researchers have improved analysis precision by combining abstract domains [130, 116, 72, 187, 31, 54, 70, 206], via the reduced product [49], reduced tensor product [148], etc. Astree [53] in particular is an industry tool that computes an approximate reduced product by propagating information through a tree of abstract domains [54]. Our approach in Concerto could be formalized as a degenerate case of this framework, where the exchange of information between the abstract and mostly-concrete interpreters takes place at the transition points via the communication channels described in [54]. However, while the reduced product domain found in Astree and other abstract interpretations typically exchange information about the same program point between multiple domains, under Concerto each program statement is analyzed by either mostly-concrete or abstract interpretation. Further, our subfixpoint iteration strategy is significantly different from the one described in [54].

Concolic Testing Our combined interpretation bears similarity to concolic testing [167, 86, 168]. Concolic testing performs symbolic and concrete execution in parallel, but falls back on concrete values in the symbolic interpreter when it encounters an expression outside of the logic of the underlying theorem prover. This approach is similar to how Concerto uses concrete execution to precisely reason about difficult-to-analyze code. Similarly, as noted in Section 6.2, our technique is similar to partial evaluation [82, 140].

However, partial evaluation is typically used for optimization [32, 98], and the full resolution of reflection is usually an orthogonal concern. We are unaware of work trying to use partial evaluation to handle difficult-to-analyze framework code for sound program analysis, and we believe CONCERTO is less brittle than a partial evaluation approach to this problem.

Handling Native Code As described in Section 7.5.2, one of the major challenges in completing our work on SYMPHONY is the handling of native methods. Many researchers have studied the problem of native code, particularly Java's native interface (JNI) [81, 80, 117, 109]. These efforts have primarily focused on finding errors in JNI code, e.g., failure to check error values or accessing nonexistent fields. Although the types inferred by [81] can help describe the fields accessed or methods called by a native method implementation, these techniques do not in general describe the behavior of JNI methods.

In [179], Tan and Morrisett present ILEA, a specification generator for JNI methods that targets a variant of the JVML language [78], a formal model of Java bytecode. Their approach is appealing as JVML can be trivially transformed into Java bytecode which could then be interpreted by JPF without any further modifications. Unfortunately, to soundly account for imprecision during model generation, their tool emits special operations which respectively encode nondeterministic selection of a value of type T, mutating an arbitrary object, and havocing the heap. Although JPF could be extended to support these operations, doing so would be either very expensive (the former two) or extremely imprecise (the latter operation).

Another drawback of ILEA is that some JNI operations are expressed in terms of other JNI operations, so ILEA is not a standalone solution for modeling all native methods. For example, JNI operations on object fields are implemented using the Java Reflection API, which itself depends on native operations. Further, the specifications created by ILEA can soundly capture the high-level heap effects of a native method, but may

miss low-level behavior critical to precision. For example, ILEA specification generation will ignore any operations that do not affect the JVM heap, including low-level I/O operations. For example, ILEA treats C's standard I/O facilities as exclusively producing nondeterministic values. As a result, specifications for the native implementations of Java's I/O APIs will themselves produce nondeterministic values. However, such specifications will be unacceptably imprecise for use in Symphony and Concerto; both projects rely on concretely reading the contents of configuration files. ILEA is thus not a complete replacement for the implementations included in JPF, but can be used for handling native methods that do not have JPF implementations.

Caching and Summary Computation Many abstract interpreters we have reviewed either do not fully report their their summary/caching strategies [71] or use (some variation on) the naïve strategy described in Section 7.5.2 [104, 57]. We are unaware of any optimizations from the abstract interpretation literature that can be directly applied to the issue of method summaries involving large concrete states. However, methods to detect redundant states developed in the random testing and symbolic execution communities may be usable in the context of Symphony [200, 194, 115]. We can potentially use these papers' techniques for detecting similarity between heap structures as a pre-filter to determine if a state is *definitely* unvisited. That is, if a state has an unseen heap shape, then it must be unvisited and more expensive value-by-value subsumption checking can be skipped.

One possible optimization discussed in Section 7.5.2 was the generation of symbolic summaries for framework code. Several researchers have investigated generating summaries of method behavior. Reps et al. have shown how to efficiently compute symbolic summaries in data-flow reachability problems [163]. The IDE framework used by Legato [166] (which generalizes the work in [163]) can generate summary functions for analyses over environments (see Section 3.3.1). However, both frameworks in [163, 166] require distributive transformers, which means they cannot fully summarize the effects

of mostly-concrete interpretation. Other researchers have developed techniques to generate symbolic summaries of a method's effect on the heap [66, 7]. Like the work in [163], these techniques can only describe the *movement* of values through the heap, not their transformation. However, SYMPHONY could use these summaries for framework methods that do not manipulate or interrogate such as setters or getters.

Chapter 9

CONCLUSION AND FUTURE WORK

In this dissertation I presented three major research efforts which use program analyses to improve the quality of highly-configurable software. The work on Staccato and Legato used static and dynamic analysis techniques to find defects in dynamic configuration updates. This work substantiated my first sub-thesis, which claims that **defects in programs support for configuration changes at runtime can be detected with static and dynamic techniques.** The work on Concerto used the rigorous theory of abstract interpretation to combine (mostly-)concrete execution and an abstract interpreter to precisely analyze applications built upon highly configurable, difficult-to-analyze frameworks. Together, the formal development and empirical evaluation of Concerto substantiated my second sub-thesis, which stated that **the performance and precision of static analyses over framework-based application can be improved with the principled combination of abstract and concrete interpretation that exploits static configuration information.** In this chapter, I briefly review the techniques developed during this research and then outline future work.

9.1 Techniques

This section highlights the key technical development for each research project presented in this dissertation.

Configuration Histories Staccato is an information flow analysis which uses the novel domain of configuration histories for its shadow state. Configuration histories record both the options and which versions of those options were used to construct a

value. These configuration histories are automatically combined as their host values are combined and transformed. Given a configuration history attached to an object, it is relatively simple to determine whether the value satisfies one of the two correctness conditions. Staccato gives high-level control over how configuration histories are transformed and propagated via propagation annotations. These annotations can express that, e.g., an entire object is derived from the configuration information present in its constructor arguments. Configuration histories are a general data structure for recording the configuration options and versions; Staccato used them to record the options and versions observed during entire method executions as a heuristic for detecting correctness violations involving control-flow.

Abstract Resource Versions An element of the ARV domain identifies a single, unique invocation of a resource access. Intuitively, an ARV at point p in the program represents an access at point q as a set of program flows leading from q to p. These flows are represented by strings of unique labels assigned to method call sites and resource accesses. In order to disambiguate between different invocations of a method call or resource access, ARVs add primes to these labels, where n primes on a label indicates the $n + 1^{\text{th}}$ most recent occurrence of that event, i.e., method call or resource access. ARVs can be compactly represented using tries that naturally encode the set of traces that identify a resource access. Although ARVs bear some similarities to the context strings used in alias analyses, the structure and application of ARVs in Legato is, to the best of my knowledge, unique.

Combined Execution Concerto analyzes framework code with mostly-concrete execution and the application code with an abstract interpreter. The mostly-concrete interpreter is simply a concrete interpreter extended to support nondeterminism and embedded abstract values. Using concrete execution to reason about programs is not new (e.g., unit testing, directed random testing, dynamic analysis), but Concerto's integra-

tion within a sound static analysis is, to the best of my knowledge, unique. In essence, Concerto optimistically assumes framework code will be completely deterministic, but automatically falls back on an imprecise answer when this assumption is violated, maintaining soundness. In general, such an optimistic assumption would be frequently violated, but frameworks are often deterministic once given concrete configuration information. Concerto exploits this configuration information to explore these deterministic paths of execution with mostly-concrete interpretation. Further, as Concerto uses an abstract interpretation on the application, this assumption does *not* need to hold for the application, *only* the framework code.

9.2 Future Directions

This dissertation represents a first step in addressing many of the challenges described in Chapter 1. This section sketches future research directions, focusing on addressing the challenges identified in Chapter 5.

Automatic Sharing of Analysis Results I propose exploring techniques for automatically sharing analysis results and facts across problem domains. Currently, results generated by one analysis generally cannot be used by another static analysis. Thus, every analysis must begin inference about program behavior from scratch, which is expensive for large programs. This lack of reuse is especially wasteful as analyses in different domains may still generate information useful to each other. For example, an array bounds analysis may compute information about points-to relations that could be reused in a taint analysis to save analysis time. I plan to research how to: 1) automatically transform static analyses to derive reusable facts about program behavior during analysis, and 2) transform other analyses to consume these facts to bootstrap inference, saving valuable developer time.

Automatic Synthesis of Weakened Analyses A common technique for static analysis is to add precision "knobs" to an analysis [91, 104, 108]. These knobs allow the analysis user to trade performance for precision. However, constructing these knobs requires careful engineering on the part of the analysis designer and implementer. I plan to research techniques to *synthesize* less precise (but more scalable) versions of existing analyses. These weakened analyses may be used as fast preanalyses, or to handle large library codebases. Existing work by Cousot et al. [55] has shown that abstract interpretation can be applied to abstract interpreters to automatically refine abstract domains and widening operators. My proposed work can be viewed as a more dramatic application of this general technique.

Sharing API Knowledge As described in Section 5.2.2, analyses rely on high-level, semantic information about method or API behavior that must be manually codified by the analysis developer. I envision developing a shared infrastructure and interchange format for static analysis researchers to share this semantic information. Designing this system and its format promises to reveal what information analysis authors use and how even further sharing can be achieved between analyses and researchers.

Symphony Usability CONCERTO requires that the integrated abstract interpreter is written against APIs that manage heap embedding, switching between the interpretation strategies, etc. These APIs enforce a relatively rigid structure on the abstract interpretation implementation, and add nontrivial code overhead. One important research direction is designing techniques for automatically transforming existing abstract interpreters to integrate SYMPHONY with as little programmer intervention as possible. A totally general transformation that integrates *any* abstract interpretation implementation into SYMPHONY is likely impossible. However, I plan to explore a lightweight system of annotations or DSLs to enable the integration of a wide variety of existing abstract interpreters with SYMPHONY.

BIBLIOGRAPHY

[1] Java class library documentation v 11. https://docs.oracle.com/en/java/javase/11/.

[2] Spring framework. http://spring.io/.

[3] T.J. Watson Libraries for Analysis (WALA). http://wala.sf.net/.

[4] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools (Second Edition)*. Addison Wesley, 2007.

[5] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *POPL*, 2016.

[6] Steven Arzt and Eric Bodden. Reviser: efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In *ICSE*, 2014.

[7] Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In *ICSE*, 2016.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

[9] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[10] Mona Attariyan and Jason Flinn. Using causality to diagnose configuration bugs. In *USENIX Annual Technical Conference*, 2008.

[11] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *OSDI*, 2010.

[12] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *ICSE*, 2014.

[13] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proceedings of the VLDB Endowment*, 8(3), 2014.

[14] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Symposium on Security and Privacy*, 2008.

[15] Paulo Barros, Suzanne Just, Renéand Millstein, Paul Vines, Werner Dietl, and Michael D Ernst. Static analysis of implicit control flow: Resolving java reflection and android intents. In *ASE*, 2015.

[16] Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *POPL*, 2015.

[17] Jonathan Bell and Gail Kaiser. Phosphor: illuminating dynamic data flow in commodity JVMs. In *OOPSLA*, 2014.

[18] Jonathan Bell and Luís Pina. Crochet: Checkpoint and rollback via lightweight heap traversal on stock jvms. In *ECOOP*, 2018.

[19] Tom Bergan, Dan Grossman, and Luis Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *OOPSLA*, 2014.

[20] Arthur J Bernstein, Philip M Lewis, and Shiyong Lu. Semantic conditions for correctness at different isolation levels. In *Data Engineering*, 2000.

[21] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Pub. Co. Inc., Reading, MA, 1987.

[22] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing systems*, 2(2), 1996.

[23] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to android framework modeling. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, 2015.

[24] Sam Blackshear and Shuvendu K. Lahiri. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *PLDI*, 2013.

[25] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *State of the Art in Java Program analysis*, 2012.

[26] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

[27] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, 1993.

[28] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, 2003.

[29] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Javaprogramming language. In *OOPSLA*.

[30] Guillaume Brat, Klaus Havelund, SeungJoon Park, and Willem Visser. Java pathfinder-second generation of a java model checker. In *Workshop on Advances in Verification*, 2000.

[31] Jörg Brauer, Thomas Noll, and Bastian Schlich. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *Workshop on Software & Compilers for Embedded Systems*, 2010.

[32] Matt Brown and Jens Palsberg. Jones-optimal partial evaluation by specialization-safe normalization. *Proc. ACM Program. Lang.*, 2(POPL):14, 2017.

[33] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *CAV*, 2011.

[34] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: A complete and automatic linearizability checker. In *PLDI*, 2010.

[35] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[36] Xiang Cai, Rucha Lale, Xincheng Zhang, and Robert Johnson. Fixing races for good: Portable and reliable unix file-system race detection. In *Information, Computer and Communications Security*, 2015.

[37] Cristiano Calcagno and Dino Distefano. Infer: an automatic program verifier for memory safety of Cprograms. In *NASA Formal Methods Symposium*, 2011.

[38] Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. Open-sourcing facebook infer: Identify bugs before you ship. https://code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/, 2015.

[39] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Compiler Construction*, 1986.

[40] Sagar Chaki, Edmund Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent c programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.

[41] Sagar Chaki, Edmund Clarke, Nicholas Kidd, Thomas Reps, and Tayssir Touili. Verifying concurrent message-passing c programs with recursive calls. In *TACAS*, 2006.

[42] Shigeru Chiba. Javassist-a reflection-based programming wizard for Java. In *OOPSLA Workshop on Reflective Programming in C++ and Java*, 1998.

[43] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[44] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for Javascript. In *PLDI*, 2009.

[45] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA*, 2007.

[46] Patrick Cousot. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. *Res. rep. RR*, 88, 1977.

[47] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[48] Patrick Cousot and Radhia Cousot. Constructive versions of tarski's fixed point theorems. *Pacific journal of Mathematics*, 82(1), 1979.

[49] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*. ACM, 1979.

[50] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2-3):103–179, 1992.

[51] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4), 1992.

[52] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *International Conference on Compiler Construction*, 2002.

[53] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *ESOP*, 2005.

[54] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In *Annual Asian Computing Science Conference*, 2006.

[55] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. A^2i: abstract^2 interpretation. *Proceedings of the ACM on Programming Languages*, 3(POPL):42, 2019.

[56] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *TOSEM*, 17(2):8, 2008.

[57] David Darais, Nicholas Labich, Phúc C Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP):12, 2017.

[58] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *CAV*, 2015.

[59] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.

[60] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.

[61] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, 2004.

[62] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), 1976.

[63] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, 1994.

[64] Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. Building and using pluggable type-checkers. In *ICSE*, 2011.

[65] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.

[66] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, 2011.

[67] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with iTuned. *Proceedings of the VLDB Endowment*, 2009.

[68] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA*, 2007.

[69] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *TOCS*, 32(2), 2014.

[70] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software*, 2010.

[71] Manuel Fahndrich and Francesco Logozzo. Static contract checking with abstract interpretation. Springer Verlag, October 2010.

[72] Pietro Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Formal Techniques for Distributed Systems*, pages 186–200. Springer, 2010.

[73] Pietro Ferrara. Generic combination of heap and value analyses in abstract interpretation. In *VMCAI*, 2014.

[74] Stephen J Fink, Eran Yahav, Nurit Dor, G Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *TOSEM*, 17(2), 2008.

[75] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, 2008.

[76] Apache Foundation. Apache struts 2. `https://struts.apache.org/`.

[77] Martin Fowler. Inversion of control containers and the dependency injection pattern. 2004.

[78] Stephen N Freund and John C Mitchell. A type system for the java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3-4), 2003.

[79] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android. Technical Report CS-TR-4991, November 2009.

[80] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *PLDI*, 2005.

[81] Michael Furr and Jeffrey S Foster. Polymorphic type inference for the jni. In *European Symposium on Programming*, pages 309–324. Springer, 2006.

[82] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[83] Malay Ganai, Dongyoon Lee, and Aarti Gupta. Dtam: dynamic taint analysis of multi-threaded programs for relevancy. In *FSE*, 2012.

[84] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1), 1987.

[85] Patrice Godefroid, Peli de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating software testing using program analysis. *IEEE software*, 25(5), 2008.

[86] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *PLDI*, 2005.

[87] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *POPL*, 2010.

[88] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: Countering unsoundness with heap snapshots. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017.

[89] Sumit Gulwani and Ashish Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP*, 2007.

[90] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Computer Security Applications Conference*, 2005.

[91] Ben Hardekopf, Ben Wiedermann, Berkeley Churchill, and Vineeth Kashyap. Widening for control-flow. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2014.

[92] Christopher M Hayden, Karla Saur, Michael Hicks, and Jeffrey S Foster. A study of dynamic software update quiescence for multithreaded programs. In *Workshop on Hot Topics in Software Upgrades*, 2012.

[93] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. In *OOPSLA*, 2012.

[94] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.

[95] Daniel Jackson. Software abstractions: Logic. *Language, and Analysis. MIT Press*, 2012, 2006.

[96] Dongpu Jin, Myra B Cohen, Xiao Qu, and Brian Robinson. PrefFinder: getting the right preference in configurable software systems. In *ASE*, 2014.

[97] Dongpu Jin, Xiao Qu, Myra B Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *ICSE*, 2014.

[98] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[99] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *POPL*, 1979.

[100] Neil D Jones and Steven S Muchnick. Flow analysis and optimization of lisp-like structures. In *POPL*, 1979.

[101] Matthias L. Jugel. Personal Communication, 2017.

[102] John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3), 1977.

[103] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[104] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: a static analysis platform for javascript. In *FSE*, 2014.

[105] Uday P Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *TOPLAS*, 30(1), 2007.

[106] Gary A Kildall. A unified approach to global program optimization. In *POPL*, 1973.

[107] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Cant live with em, cant live without em. In *Information Systems Security*, 2008.

[108] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. Practically tunable static analysis framework for large-scale javascript applications (t). In *ASE*, 2015.

[109] Goh Kondoh and Tamiya Onodera. Finding bugs in java native interface programs. In *ISSTA*, 2008.

[110] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. Sound static deadlock analysis for c/pthreads. In *ASE*, 2016.

[111] Eugene Kuleshov. Using the ASMframework to implement common Javabytecode transformation patterns. *Aspect-Oriented Software Development*, 2007.

[112] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. Accelerating program analyses by cross-program training. In *OOPSLA*, 2016.

[113] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In *Partial Evaluation and Semantics-based Program Manipulation*, 2008.

[114] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE software*, 21(3), 2004.

[115] Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *ICSE*, 2008.

[116] Vincent Laviron and Francesco Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI*, 2009.

[117] Byeongcheol Lee, Ben Wiedermann, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Jinn: Synthesizing dynamic bug detectors for foreign language interfaces. In *PLDI*, 2010.

[118] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths. In *ASE*, 2015.

[119] Yue Li, Tian Tan, and Jingling Xue. Effective soundness-guided reflection analysis. In *SAS*, 2015.

[120] Max Lillack, Christian Kästner, and Eric Bodden. Tracking load-time configuration options. In *ASE*, 2014.

[121] Yu Lin. *Automated refactoring for Java concurrency*. PhD thesis, University of Illinois at Urbana-Champaign, 2015.

[122] Yu Lin and Danny Dig. Check-then-act misuse of Javaconcurrent collections. In *ICST*, 2013.

[123] Mario Linares-Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. Documenting database usages and schema constraints in database-centric applications. In *ISSTA*, 2016.

[124] Peng Liu, Omer Tripp, and Xiangyu Zhang. Flint: fixing linearizability violations. In *OOPLSA*, 2014.

[125] Ben Livshits. Standford securibench suite. `http://suif.stanford.edu/~livshits/securibench/`, 2017.

[126] Benjamin Livshits. *Improving software security with precise static and runtime analysis*. PhD thesis, Stanford University, 2006.

[127] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. In *PLDI*, 2009.

[128] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, JoséNelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2), 2015.

[129] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. Technical report, Stanford University, August 2005.

[130] Francesco Logozzo and Manuel Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Symposium on Applied Computing*, 2008.

[131] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, 2014.

[132] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[133] Kristis Makris and Rida A Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *USENIX Annual Technical Conference*, 2009.

[134] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys*, 2007.

[135] Jeremy Manson, William Pugh, and Sarita V Adve. *The Javamemory model*, volume 40. ACM, 2005.

[136] Andy Maule, Wolfgang Emmerich, and David S Rosenblum. Impact analysis of database schema changes. In *ICSE*, 2008.

[137] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In *FSE*, 2013.

[138] William S. McPhee. Operating system integrity in os/vs2. *IBM Systems Journal*, 13(3), 1974.

[139] Jerome Miecznikowski and Laurie Hendren. Decompiling java using staged encapsulation. In *Working Conference on Reverse Engineering*, 2001.

[140] Torben Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1995.

[141] Rajiv Mordani and Shing Wai Chan. Java servlet specification. 2009.

[142] Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient incremental static analysis using path abstraction. In *FASE*, 2014.

[143] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, 2014.

[144] Nomair A. Naeem and Ondrej Lhotak. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, 2008.

[145] Mayur Hiru Naik. *Effective Static Race Detection For Java*. PhD thesis, Stanford, 2008.

[146] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *PLDI*, 2006.

[147] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, 2002.

[148] Flemming Nielson. Tensor products generalize the relational data flow analysis method. In *4th Hungarian Computer Science Conference*, pages 211–225, 1985.

[149] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA*, 2002.

[150] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *ISSTA*, 2008.

[151] Mathias Payer and Thomas R. Gross. Protecting applications against tocttou races by user-space caching of file metadata. In *VEE*, 2012.

[152] Mathias Payer and Thomas R. Gross. Protecting applications against TOCTTOUraces by user-space caching of file metadata. In *VEE*, 2012.

[153] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: DSUfor Javaon a stock JVM. In *OOPSLA*, 2014.

[154] Lori L. Pollock and Mary Lou Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12), 1989.

[155] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. Javadaptorflexible runtime updates of Javaapplications. *Software: Practice and Experience*, 2013.

[156] Dong Qiu, Bixin Li, and Zhendong Su. An empirical analysis of the co-evolution of schema and code in database applications. In *FSE*, 2013.

[157] Ariel Rabkin and Randy Katz. Precomputing possible configuration error diagnoses. In *ASE*, 2011.

[158] Ariel Rabkin and Randy Katz. Static extraction of program configuration options. In *ICSE*, 2011.

[159] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *PLDI*, 2007.

[160] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for java libraries. In *FSE*, 2016.

[161] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE*, 2010.

[162] Brianna M. Ren and Jeffrey S. Foster. Just-in-time static type checking for dynamic languages. In *PLDI*, 2016.

[163] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[164] Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *International Conference on Compiler Construction*, 2008.

[165] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, 2015.

[166] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2), 1996.

[167] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV*, 2006.

[168] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[169] Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, 2011.

[170] Ohad Shacham, Eran Yahav, Guy Golan Gueta, Alex Aiken, Nathan Bronson, Mooly Sagiv, and Martin Vechev. Verifying atomicity via data independence. In *ISSTA*, 2014.

[171] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of java reflection. In *ASPLAS*, 2015.

[172] Amie L. Souter and Lori L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *ICSM*, 2001.

[173] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. 2016.

[174] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. In *OOPSLA*, 2011.

[175] Robert E Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, (1), 1986.

[176] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. Autobash: Improving configuration management with operating system causality analysis. In *SOSP*, 2007.

[177] Gregory T Sullivan. Dynamic partial evaluation. In *Programs as Data Objects*, pages 238–256. 2001.

[178] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *OOPSLA*, 2000.

[179] Gang Tan and Greg Morrisett. Ilea: Inter-language analysis across java and c. In *OOPSLA*, 2007.

[180] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2), 1955.

[181] Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Deep typechecking and refactoring. In *OOPSLA*, 2008.

[182] John Toman and Dan Grossman. Staccato: A Bug Finder for Dynamic Configuration Updates. In *ECOOP*, 2016.

[183] John Toman and Dan Grossman. Taming the static analysis beast. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 71, 2017.

[184] John Toman and Dan Grossman. Legato: An at-most-once analysis with applications to dynamic configuration updates. In *ECOOP*, 2018.

[185] John Toman and Dan Grossman. Concerto: a framework for combined concrete and abstract interpretation. *PACMPL*, 3(POPL):43, 2019.

[186] Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *ICSE*, 2010.

[187] Antoine Toubhans, Bor-Yuh Evan Chang, and Xavier Rival. Reduced product combination of abstract domains for shapes. In *VMCAI*, 2013.

[188] Jesse A. Tov and Riccardo Pucella. Practical affine types. In *POPL*, 2011.

[189] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *PLDI*, 2009.

[190] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *PLDI*, 2009.

[191] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, 1999.

[192] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction*, 2000.

[193] David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP*, 2010.

[194] Willem Visser, Corina S Păsăreanu, and Radek Pelánek. Test input generation for java containers using state matching. In *ISSTA*, 2006.

[195] Philip Wadler. Linear types can change the world. In *IFIP TC*, 1990.

[196] Liqiang Wang and Scott D Stoller. Runtime analysis of atomicity for multithreaded programs. *Transactions on Software Engineering*, 2006.

[197] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J Wang, Chun Yuan, and Zheng Zhang. Strider: A black-box, state-based approach to change and configuration management and support. *Science of Computer Programming*, 2004.

[198] Mark N Wegman and F Kenneth Zadeck. Constant propagation with conditional branches. *TOPLAS*, 13(2), 1991.

[199] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *ISSTA*, 2013.

[200] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2005.

[201] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating range fixes for software configuration. In *ICSE*, 2012.

[202] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *FSE*, 2015.

[203] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *SOSP*, 2013.

[204] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *ICSE*, 2015.

[205] Daniel M Yellin and Robert E Strom. Protocol specifications and component adaptors. *TOPLAS*, 19(2), 1997.

[206] Matteo Zanioli, Pietro Ferrara, and Agostino Cortesi. Sails: Static analysis of information leakage with sample. In *Symposium on Applied Computing*, 2012.

[207] Sai Zhang and Michael D Ernst. Automated diagnosis of software configuration errors. In *ICSE*, 2013.

[208] Sai Zhang and Michael D Ernst. Which configuration option should Ichange? In *ICSE*, 2014.

[209] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid top-down and bottom-up interprocedural analysis. In *PLDI*, 2014.

[210] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

[211] Haiyan Zhu, Thomas Dillig, and Isil Dillig. Automated inference of library specifications for source-sink property verification. In *ASPLAS*, 2013.

Appendix A

PROOFS FOR CHAPTER 3

A.1 Preliminaries

Although the environment transformers presented in Chapter 3 gave semantics as denotations from statements to environment transformers, the IDE framework of Sagiv et al. assigns transformers to edges in the program control flow graph. Following the notation of Sagiv et al. in [166], assume we have a function $M : E^* \rightarrow (Env \rightarrow Env)$, which maps an edge in the program control-flow graph to an environment transformer. This function naturally extends to paths of edges by composing the environment transformers for each successive edge in a path.

The solution computed by the IDE framework is the meet-over-all-paths solution,[1] defined for a distinguished start node s_0 and start environment Ω as:

$$MOP(n) \triangleq \bigsqcap_{p \in path(s_0, n)} M(p)(\Omega)$$

In other words, the meet-over-all-paths the meet of applying the transformers for every path from s_0 to n to the start environment Ω.

Our proofs exploit this path-based paradigm: we give the abstract and concrete instrumented semantics as assignments of transformers to edges. It is easy to see the correspondence to the transformers presented in Chapter 3.

[1]Technically, when considering interprocedural programs, the IDE framework computes the meet-over-all-*valid*-paths solution. As we do not consider methods in this section, we instead state our proofs using the simpler notion of meet-over-all-paths.

A.2 Concrete Instrumented Semantics

We first define the domain of concrete instrumented states as: $S = (X \to \mathbb{P}(\mathbb{N})) \times \mathbb{N}$, where X is the finite domain of variables that appear in a given program. We denote a concrete instrumented state of type S with $\langle env, c \rangle$. The instrumented semantics are given by the following assignment of transformers of type $S \to S$ to edges in the program supergraph:

$$\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \to s_1 \triangleq id \tag{A.1}$$

$$\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \to s_2 \triangleq id \tag{A.2}$$

$$\texttt{while } e \texttt{ do } s \texttt{ end} \to s \triangleq id \tag{A.3}$$

$$\texttt{while } e \texttt{ do } s \texttt{ end} \to s' \triangleq id \tag{A.4}$$

$$\texttt{skip} \to s \triangleq id \tag{A.5}$$

$$x = y \to s \triangleq \lambda \langle env, c \rangle . \langle env[x \mapsto env[y]], c \rangle \tag{A.6}$$

$$x = c \to s \triangleq \lambda \langle env, c \rangle . \langle env[x \mapsto \emptyset], c \rangle \tag{A.7}$$

$$x = y + z \to s \triangleq \lambda \langle env, c \rangle . \langle env[x \mapsto env[y] \cup env[z]], c \rangle \tag{A.8}$$

$$x = \texttt{get}^{\ell}\texttt{()} \to s \triangleq \lambda \langle env, c \rangle . \langle env[x \mapsto \{c\}], c + 1 \rangle \tag{A.9}$$

Where the edge in Equation (A.1) refers to the edge from the conditional header to the node corresponding to the branch statement s_1, and similarly for Equation (A.2) and the false branch s_2. The edge in Equation (A.3) corresponds to when the loop condition is true, and the loop body executed, whereas the edge in Equation (A.4) is when the loop condition is false and the loop is skipped. All other edges refer to the (unique) edge from a statement to its successor in the supergraph.

Define $C(p)$ as the composition of the transformers corresponding to each edge in the path p. Let Ω be the initial instrumented state, defined to be: $\langle \lambda_.\emptyset, 0 \rangle$.

A.3 Abstract Semantics

Let the domain of primed labels be denoted by $L = \widehat{\ell}^n \cup \{\top, \bot\}$. The environments used in the paper are of type $\widehat{S} = X \to L$. We will denote environments of type $\widehat{S}$ with $\widehat{env}$. The distributive environment transformers in Sections 3.3.2 and 3.3.3 are equivalent to the transformers of type $\widehat{S} \to \widehat{S}$ assigned to the edges in the supergraph:

$$\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \to s_1 \triangleq \mathrm{id} \tag{A.10}$$

$$\texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \to s_2 \triangleq \mathrm{id} \tag{A.11}$$

$$\texttt{while } e \texttt{ do } s \texttt{ end} \to s \triangleq \mathrm{id} \tag{A.12}$$

$$\texttt{while } e \texttt{ do } s \texttt{ end} \to s' \triangleq \mathrm{id} \tag{A.13}$$

$$\texttt{skip} \to s \triangleq \mathrm{id} \tag{A.14}$$

$$x = y \to s \triangleq \lambda \widehat{env}.\widehat{env}[x \mapsto \widehat{env}[y]] \tag{A.15}$$

$$x = c \to s \triangleq \lambda \widehat{env}.\widehat{env}[x \mapsto \top] \tag{A.16}$$

$$x = y + z \to s \triangleq \lambda \widehat{env}.\widehat{env}[x \mapsto \widehat{env}[y] \sqcap \widehat{env}[z]] \tag{A.17}$$

$$x = \texttt{get}^{\ell}() \to s \triangleq \lambda env.\lambda v. \begin{cases} \widehat{\ell} & \text{if } v = v' \\ \widehat{\ell}^{n+1} & \text{if } env(v') = \widehat{\ell}^n \\ env(v') & \text{o.w.} \end{cases} \tag{A.18}$$

Where the edges have the same interpretation as those given for the concrete semantics. At first glance, the use of id for loops and conditionals may appear incorrect. However, because the IDE framework computes the meet over all paths solution, the final result of the analysis takes the meet of all paths through a conditional, giving us the same effect. A similar observation applies for computing loop fixpoints.

Let $A(p)$ be the composition of the environment transformers corresponding to each edge in the path p, and let the initial abstract state $\widehat{\Omega}$ be defined to be $\top_{\widehat{S}}$, i.e., an environment that maps all variables to $\top$.

A.4 Proof

Define the invariant relation for two states as follows, $\langle env, c\rangle \sim \widehat{env}$ iff the following conditions hold:

$$\forall x.|env[x]| > 1 \Rightarrow \widehat{env}[x] = \bot \tag{Invariant 1}$$

$$\forall x, y, m, n.m \neq n \wedge env[x] = \{m\} \wedge env[y] = \{n\} \Rightarrow \widehat{env}[x] \neq \widehat{env}[y] \vee \widehat{env}[x] = \bot \vee \widehat{env}[y] = \bot \tag{Invariant 2}$$

$$\forall x.env[x] \neq \emptyset \Leftrightarrow \widehat{env}[x] \neq \top \tag{Invariant 3}$$

We now show that:

Theorem 7.

$$\forall n, n', p, p'.p' \equiv p \circ n \circ n' \wedge p' \in path(s, n') \wedge C(p \circ n)(\Omega) \sim A(p \circ n)(\widehat{\Omega})$$
$$\Rightarrow C(p')(\Omega) \sim A(p')(\widehat{\Omega})$$

Theorem 7 states that if the invariant holds for the two environments yielded by the transformers along the path $p \circ n$, the invariant still holds after applying the respective transformers for the edge $n \to n'$.

Proof. Let $C(p \circ n)(\Omega) = \langle env, c\rangle$ and $A(p \circ n)(\widehat{\Omega}) = \widehat{env}$, and let $C(p')(\Omega) = \langle env', c'\rangle$ and $A(p')(\widehat{\Omega}) = \widehat{env}'$. We assume $\langle env, c\rangle \sim \widehat{env}$ and must show that $\langle env', c'\rangle \sim \widehat{env}'$. We proceed on the type of edge $n \to n'$ that makes up the final component of the path p'.

Cases (A.1), (A.2), (A.3), (A.4), (A.5), (A.6), (A.7): Trivial

Case (A.8): It suffices to show that after executing the environment transformer all invariants hold for the variable x on the left-hand side of the assignment.

Invariant 1 If $|env[y]| > 1$ or $|env[z]| > 1$ then by definition of $\sim$, $\widehat{env}[y] = \bot$ or $\widehat{env}[z] = \bot$, and by the definition of meet, $\widehat{env}'[x] = \widehat{env}[y] \sqcap \widehat{env}[z] = \bot$,

preserving the invariant. Consider the case now where $|env[y]| = 1 \wedge |env[z]| = 1 \wedge env[y] \neq env[z]$. Then by the definition of $\sim$, $\widehat{env}[y] \neq \widehat{env}[z]$ or one or both of $\widehat{env}[y]$ and $\widehat{env}[z]$ is $\bot$. In either case, $\widehat{env}'[x] = \widehat{env}[y] \sqcap \widehat{env}[z] = \bot$, again preserving the invariant.

Invariant 2 If $env'[x] = \{m\} = env[y] \cup env[z]$, then either:

1. $env[y] = \{m\}$ and $env[z] = \{m\}$. Then by invariant 3, we have that $\widehat{env}[y] \neq \top$ and $\widehat{env}[z] \neq \top$. If either $\widehat{env}[y]$ or $\widehat{env}[z]$ is $\bot$, then $\widehat{env}'[x] = \bot$ and the condition is trivially satisfied. Similarly, if $\widehat{env}[y]$ and $\widehat{env}[z]$ are distinct, non-$\bot$ values, then $\widehat{env}'[x]$ will be $\bot$ and again the invariant is trivially satisfied. Finally, consider the case where $\widehat{env}[y] = \widehat{env}[z]$. Then $\widehat{env}'[x] = \widehat{env}[y] = \widehat{env}[z]$, and thus the invariant must hold by transitivity of equality and the invariant relation on the input environments.
2. $env[y] = \{m\}$ and $env[z] = \emptyset$. Then invariant 3 implies that $\widehat{env}[y] \neq \top$ and $\widehat{env}[z] = \top$, whence $\widehat{env}'[x] = \widehat{env}[y]$. If $\widehat{env}[y] = \bot$ then the invariant is trivially satisfied, otherwise the invariant holds from the transitivity of equality and the invariant on the input environments.
3. $env[y] = \emptyset$ and $env[z] = \{m\}$ follows from symmetric reasoning to the above.

Invariant 3 If $env'[x] \neq \emptyset$, then $env[y] \neq \emptyset \vee env[z] \neq \emptyset$. By invariant 3 on the input environments, this implies that $\widehat{env}[y] \neq \top \vee \widehat{env}[z] \neq \top$. By the definition of meet, we must have $\widehat{env}'[x] \neq \top$ as required.

To establish the other direction of the bi-implication, it suffices to show that $env'[x] = \emptyset \Rightarrow \widehat{env}'[x] = \top$. If $env'[x] = \emptyset$, then $env[y] = \emptyset \wedge env[z] = \emptyset$, whence by the invariant on the input environments, we have $\widehat{env}[y] = \top \wedge \widehat{env}[y] = \top$. As $\top \sqcap \top = \top$, we have the desired result.

Case (A.9): We again establish the invariants post assignment.

Invariants 1 and 3 Trivial.

Invariant 2 By simple proof by contradiction, it can be shown that c is greater than any version number that appears in env. Thus, as $env'[x] = \{c\}$ is distinct from all other singleton version sets, it suffices to show that $\widehat{env}'[x]$ is likewise distinct from all other abstract versions. As the environment transformer in (A.18) adds a prime to existing values of the form $\widehat{\ell}^n$, this ensures that $\widehat{env}'[x] = \widehat{\ell}$ is unique within $\widehat{env}'$. Finally, for $y \neq x$, the priming process preserves inequality between abstract resource versions, ensuring the invariant holds.

□

Corollary 7.1. $\forall n, p. p \in path(s, n) \Rightarrow C(p)(\Omega) \sim A(p)(\widehat{\Omega})$

Proof. By straightforward induction on path length and application of Theorem 7. □

We can now state the main soundness result:

Theorem 8. $\forall p, n, x. p \in path(s, n) \wedge |C(p)(\Omega)[x]| > 1 \Rightarrow \left[\bigsqcap_{q \in path(s,n)} A(q)(\widehat{\Omega})\right][x] = \bot$

In other words, Theorem 8 states that if any execution, at some point in the program a variable is derived from multiple versions of the resource, the analysis derives $\bot$ for that variable at that point.

Proof. Observe that if, for some x, $|C(p)(\Omega)[x]| > 1$, then $A(p)(\widehat{\Omega})[x] = \bot$ by Corollary 7.1, and by the definition of meet, $\bigsqcap_{q \in path(s,n)} A(q)(\widehat{\Omega})[x] = \bot$ □

Appendix B

PROOFS FOR CHAPTER 6

B.1 Proofs for Section 6.4

B.1.1 Soundness of $I_\top$

We now prove that $\alpha_F \circ F \sqsubseteq I_\top \circ \alpha_F$. We first note the following fact about α_F that we will exploit during our proofs:

$$\forall \vec{\ell}, \vec{\ell}', r, r', x, y. \mathcal{V}(r[\vec{\ell}], x) \subseteq \mathcal{V}(r'[\vec{\ell}], y) \Rightarrow \alpha_F(r)[\vec{\ell}][x] \sqsubseteq \alpha_F(r')[\vec{\ell}'][y] \tag{B.1}$$

$$\alpha_v(\emptyset) = \bot_{\widehat{A}} \tag{B.2}$$

To begin, we will ignore the "initial state" term from the definition of F, and first prove $I_\top$ sound with respect to:

$$F_0(r)[\vec{\ell}] = \begin{cases} \bigcup_{\substack{p \in pred(\ell) \\ in \in r[p^\bullet \rightsquigarrow \ell^\circ]}} step^F(in, \ell) & \vec{\ell} = s^\circ \rightsquigarrow s^\bullet \\ F(r)[\vec{\ell}] & o.w. \end{cases}$$

which is the original concrete semantic function without the intialization term. We first prove that the inequality holds for some arbitrary r and $\ell^\circ \rightsquigarrow \ell^\bullet$. We first need the following lemmas.

Lemma 1. $\forall \vec{\ell}, r, \widetilde{s}. \alpha_F([\vec{\ell} \mapsto \mathcal{FL}])[\vec{\ell}] \sqsubseteq \widetilde{s} \Rightarrow \alpha_F \circ F_0(r)[\vec{\ell}] \sqsubseteq \widetilde{s}$ *where* $\mathcal{FL} = F_0(r)[\vec{\ell}]$ *and* $[\vec{\ell} \mapsto s] : R$ *is shorthand for* $\bot_R[\vec{\ell} \mapsto s]$.

Proof. For every x, $\mathcal{V}(F_0(r)[\vec{\ell}], x) = \mathcal{V}([\vec{\ell} \mapsto \mathcal{FL}][\vec{\ell}], x)$, and thus by (B.1), $\alpha_F([\vec{\ell} \mapsto \mathcal{FL}])[\vec{\ell}][x] = \alpha_F(F_0(r))[\vec{\ell}][x]$. By transitivity, we have $\alpha_F \circ F_0(r)[\vec{\ell}] \sqsubseteq \widetilde{s}$. □

Intuitively, Lemma 1 ensures that to establish pointwise inequality, it suffices to consider the abstracted state for each $\vec{\ell}$ individually.

Lemma 2.

$$\begin{aligned}&\forall r, \ell, p \in \mathit{pred}(\ell), \widetilde{s}.\\&\quad \big(\forall \langle in, E\rangle \in r[p^{\bullet} \rightsquigarrow \ell^{\circ}].\alpha_F([\ell \mapsto \mathit{step}^F(\langle in, E\rangle, \ell)])[\ell] \sqsubseteq \widetilde{s}\big)\\&\quad \Rightarrow \alpha_F([\ell \mapsto \mathcal{FL}])[\ell] \sqsubseteq \widetilde{s}\end{aligned}$$

where $\mathcal{FL}$ is defined as in Lemma 1.

Proof. First, we show that $\bot_{\widetilde{R}}[\ell \mapsto \widetilde{s}]$ is an upper bound for the set $\{\alpha_F(m) \mid m \in \mathcal{M}\}$, where $\mathcal{M} = \{[\ell \mapsto \mathit{step}^F(\langle in, E\rangle, \ell)] \mid p \in pred(\ell) \wedge \langle in, E\rangle \in r[p^{\bullet} \rightsquigarrow \ell^{\circ}]\}$. Let m be some element of $\mathcal{M}$. By assumption, $\alpha_F(m)[\ell] \sqsubseteq \widetilde{s}$, so it remains to show that $\forall \ell'.\alpha_F(m)[\ell'] \sqsubseteq \bot_{\widetilde{R}}[\ell \mapsto \widetilde{s}][\ell'] = \bot_{\widetilde{S}}$. This follows from the fact that for any x, $\alpha_F(\bot_R)[\ell'][x] = \bot_{\widetilde{V}}$ (where $\bot_{\widetilde{V}}$ is $\bot_{\widehat{A}}$ or $\emptyset$, depending on the type of x) and by (B.1) and $\mathcal{V}(\bot_R[\ell'], x) = \emptyset = \mathcal{V}(m[\ell'], x)$, whence we have that $\alpha_F(m)[\ell'][x] = \bot_{\widetilde{V}}$.

Next, observe that $[\ell \mapsto \mathcal{FL}] = [\ell \mapsto \bigcup_{\substack{p \in \mathit{pred}(\ell)\\ \langle in,E\rangle \in r[p^{\bullet} \rightsquigarrow \ell^{\circ}]}} \mathit{step}^F(\langle in, E\rangle, \ell)]$ which is equivalent to $\bigsqcup_{\substack{p \in \mathit{pred}(\ell)\\ \langle in,E\rangle \in r[p^{\bullet} \rightsquigarrow \ell^{\circ}]}} [\ell \mapsto \mathit{step}^F(\langle in, E\rangle, \ell)]$, i.e., $\bigsqcup \mathcal{M}$. As α_F preserves least upper bounds, and from the fact that $\bot_{\widetilde{R}}[\ell \mapsto \widetilde{s}]$ is an upper bound of the set $\{\alpha_F(m) \mid m \in \mathcal{M}\}$, we have that: $\alpha_F(\bigsqcup_{\substack{p \in \mathit{pred}(\ell)\\ \langle in,E\rangle \in r[p^{\bullet} \rightsquigarrow \ell^{\circ}]}} [\ell \mapsto \mathit{step}^F(\langle in, E\rangle, \ell)]) \sqsubseteq \bot_{\widetilde{R}}[\ell \mapsto \widetilde{s}]$, from which it is immediate that $\alpha_F([\ell \mapsto \mathcal{FL}])[\ell] \sqsubseteq \widetilde{s}$. □

Lemma 2 implies that to show an abstract state $\widetilde{s}$ over-approximates the result of stepping all incoming concrete states, it suffices to show that $\widetilde{s}$ over-approximates stepping each individual incoming state. We now show that for any ℓ and r, $I_{\top}(\alpha_F(r))[\ell^{\circ} \rightsquigarrow \ell^{\bullet}]$ is such an $\widetilde{s}$.

Lemma 3. $\forall r, \ell, in, p \in \mathit{pred}(\ell).\langle in, E\rangle \in r[p^{\bullet} \rightsquigarrow \ell^{\circ}]$:

1.

$$\begin{aligned}&\forall x, y.\mathcal{V}(\mathit{step}^F(\langle in, E\rangle, \ell), x) \subseteq \mathcal{V}(\{\langle in, E\rangle\}, y)\\&\Rightarrow \alpha_F([\ell \mapsto \mathit{step}^F(\langle in, E\rangle, \ell)])[\ell][x] \sqsubseteq \Big(\bigsqcup_{p' \in pred(\ell)} \alpha_F(r)[p'^{\bullet} \rightsquigarrow \ell^{\circ}]\Big)[y]\end{aligned}$$

2. $\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell] \sqsubseteq I_\top(\alpha_F(r))[\ell]$

Proof.

1. $\langle in, E\rangle \in r[p^\bullet \rightsquigarrow \ell^\circ]$ implies that $\mathcal{V}(\{\langle in, E\rangle\}, y) \subseteq \mathcal{V}(r[p^\bullet \rightsquigarrow \ell^\circ], y)$, whence by transitivity and (B.1) we then have:

$$\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][x] \sqsubseteq \alpha_F(r)[p^\bullet \rightsquigarrow \ell^\circ][y]$$

   Transivity and least upper bounds gives

$$\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][x] \sqsubseteq \alpha_F(r)[p^\bullet \rightsquigarrow \ell^\circ][y] \sqsubseteq \bigsqcup_{p' \in pred(\ell)} \alpha_F(r)[p'^\bullet \rightsquigarrow \ell^\circ][y]$$

2. By the pointwise definition of the domains, it suffices to show that:

$$\begin{aligned} \forall x. \alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][x] &\sqsubseteq step^\top(\widetilde{in}, \ell)[x] \\ &\sqsubseteq I_\top(\alpha_F(r))[\ell][x] \end{aligned}$$

   where $\widetilde{in} = \bigsqcup_{p' \in pred(\ell)} \alpha_F(r)[p'^\bullet \rightsquigarrow \ell^\circ]$. As an immediate consequence of part 1 above, we can conclude that if all states produced by $step^F$ retain the original value of x, and that $step^\top$ likewise does not change the value of x from $\widetilde{in}$, the inequality holds at x.

   We proceed by the branches of $step^\top$:

   Case (6.5): $step^\top$ simply returns $\widetilde{in}$, and $step^F(\langle in, E\rangle, \ell)$ must return $\{\langle in, E\rangle\}$ and by the above argument, the inequality holds for all x.

   Case (6.6): We must only establish the inequality for the left-hand side of the assignment. As $\mathcal{V}(step^F(\langle in, E\rangle, \ell), lhs) = \mathcal{V}(\{\langle in, E\rangle\}, rhs)$, we have by part (1) above:

$$\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][lhs] \sqsubseteq \widetilde{in}[rhs] = step^\top(\widetilde{in}, \ell)[lhs]$$

Case (6.7): As above, we need only establish that the relationship holds for the lhs. By definition of $step^F$, $\mathcal{V}(step^F(\langle in, E\rangle, \ell), lhs) = \mathcal{V}(\{\langle in[lhs \mapsto [\![bc_f]\!]], E\rangle\}, lhs) = \{[\![bc_f]\!]\}$. By the definition of α_F:

$$\begin{aligned}\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][lhs] &= \{[\![bc_f]\!]\} \\ &= \widetilde{in}[lhs \mapsto \{[\![bc_f]\!]\}][lhs] \\ &= step^\top(\widetilde{in}, \ell)[lhs]\end{aligned}$$

Case (6.8): Except for the lhs, every variable in each state produced by $step^F$ retains its original value in in, and thus we must only show that the inequality holds for lhs. Define the set $\mathcal{O} = \{r \mid \langle r, E'\rangle \in [\![fop]\!](E, in[v_1], \ldots, in[v_n]\}$ where $v_1, \ldots, v_n$ are the variable arguments to the *fop*. By the definition of α_F, $\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][lhs] = \mathcal{O}$. Consider now the value of $step^\top(\widetilde{in}, \ell)[lhs]$. If it is $\top_F$, then the result trivially again holds. Otherwise, we can again conclude that $step^\top(\widetilde{in}, \ell)[lhs] = \widetilde{[\![fop]\!]}(\widetilde{in}[v_1], \ldots, \widetilde{in}[v_n]) \neq \top_F$, whence by the the definition of $\widetilde{[\![fop]\!]}$, we may further conclude that $\widetilde{in}[v_1], \ldots, \widetilde{in}[v_n] \neq \top_F$. By the definition of least upper bounds and from the definition of α_F, we must then have: $in[v_1] \in \widetilde{in}[v_1] \wedge \ldots \wedge in[v_n] \in \widetilde{in}[v_n]$. As $E \in \mathcal{E}$, from the definition of $\widetilde{[\![fop]\!]}$ we have $\mathcal{O} \subseteq \widetilde{[\![fop]\!]}(\widetilde{in}[v_1], \ldots, \widetilde{in}[v_n]) = step^\top(\widetilde{in}, \ell)[lhs]$ as required.

Case (6.9): By definition of α_F, $\alpha_F([\ell \mapsto step^F(\langle in, E\rangle, \ell)])[\ell][lhs] = \hat{v} : \hat{A} \sqsubseteq \top_{\hat{A}} = step^\top(\widetilde{in}, \ell)[lhs]$.

□

Let us now prove soundness of $I_\top$ w.r.t F_0 for $\ell^\circ \rightsquigarrow \ell^\bullet$:

Lemma 4. $\forall \ell, r. \alpha_F \circ F_0(r)[\ell^\circ \rightsquigarrow \ell^\bullet] \sqsubseteq I_\top \circ \alpha_F(r)[\ell^\circ \rightsquigarrow \ell^\bullet]$

Proof. Immediate from Lemmas 1 and 2, and Lemma 3 part 2. □

Lemma 5. $\forall p^\bullet \rightsquigarrow \ell^\circ, r. \alpha_F \circ F_0(r)[p^\bullet \rightsquigarrow \ell^\circ] \sqsubseteq I_\top \circ \alpha_F(r)[p^\bullet \rightsquigarrow \ell^\circ]$

Proof. As $\alpha_F(\bot_R) = \bot_{\widetilde{R}}$, if $F_0(r)[p^\bullet \leadsto \ell^\circ] = \emptyset$ the inequality trivially holds. Otherwise, by definition, $F_0(r)[p^\bullet \leadsto \ell^\circ] \subseteq r[p]$, and thus $\alpha_F(F_0(r))[p^\bullet \leadsto \ell^\circ] \sqsubseteq \alpha_F(r)[p]$ (where p is shorthand for $p^\circ \leadsto p^\bullet$). If $\widetilde{\mathcal{FT}}(\alpha_F(r), p^\bullet \leadsto \ell^\circ)$ is true, then $I_\top(\alpha_F(r))[p^\bullet \leadsto \ell^\circ] = \alpha_F(r)[p]$ as required. It therefore suffices to show that $F_0(r)[p^\bullet \leadsto \ell^\circ] \neq \emptyset \Rightarrow \widetilde{\mathcal{FT}}(\alpha_F(r), p^\bullet \leadsto \ell^\circ)$. We proceed by cases on the form of $\mathtt{prog}[p]$. If $p \in \ell_f$ and $\mathtt{prog}[p]$ is conditional over x and y with target ℓ, then there exists some state $s \in r[p]$ such that $s[x][\![\texttt{<=>}]\!]s[y]$. If either $\alpha_F(r)[p][x] = \top_F$ or $\alpha_F(r)[p][y] = \top_F$, then by definition $\alpha_F(r)[p][x]\widetilde{[\![\texttt{<=>}]\!]}\alpha_F(r)[p][y]$ is trivially true. Otherwise, by the definition of α_F, $s[x] \in \alpha_F(r)[p][x]$ and similarly for $s[y]$ and $\alpha_F(r)[p][y]$, thus $\alpha_F(r)[p][x]\widetilde{[\![\texttt{<=>}]\!]}\alpha_F(r)[p][y] \Rightarrow \widetilde{\mathcal{FT}}(\alpha_F(r), p^\bullet \leadsto \ell^\circ)$. A similar argument holds for when $p \in \ell_f$, and $\mathtt{prog}[p]$ is a condition with fallthrough target ℓ. Finally, for any other statement, $\widetilde{\mathcal{FT}}$ is trivially true. □

We can now prove our main result:

Proof of Theorem 4. For $\vec{\ell} \neq s^\circ \leadsto s^\bullet$, $\alpha_F \circ F_0$ is equivalent to $\alpha_F \circ F$, whence the result holds from Lemmas 4 and 5. It remains to show that the proof holds at $\vec{\ell} = s^\circ \leadsto s^\bullet$ (which we will abbreviate in the following as s).

We must show that, for some arbitrary r, $\alpha_F(F(r))[s] \sqsubseteq I_\top(\alpha_F(r))[s]$. By reasoning similar to Lemma 1, it suffices to show:

$$\alpha_F([s \mapsto \bigcup_{\substack{p \in \mathit{pred}(\ell) \\ in \in r[p^\bullet \leadsto \ell^\circ]}} \mathit{step}^F(in, \ell)] \sqcup [s \mapsto \bigcup_{e \in \iota_{\mathcal{E}}} \mathit{step}^F(\langle \iota_S, e\rangle, \ell)]) \sqsubseteq I_\top(\alpha_F(r))$$

As α_F is a complete join morphism, this is equivalent to showing that:

$$\alpha_F([s \mapsto F_0(r)[\vec{\ell}]]) \sqcup \alpha_F([s \mapsto \bigcup_{e \in \iota_{\mathcal{E}}} \mathit{step}^F(\langle \iota_S, e\rangle, \ell)]) \sqsubseteq I_\top(\alpha_F(r))$$

By Lemmas 2 and 3, we have the necessary bound for the first term of the join. It therefore remains to show that:

$$\alpha_F([s \mapsto \bigcup_{e \in \iota_{\mathcal{E}}} \mathit{step}^F(\langle \iota_S, e\rangle, \ell)]) \sqsubseteq I_\top(\alpha_F(r))$$

for which it suffices to to show that:

$$\alpha_F([s \mapsto \bigcup_{e \in \iota_{\mathcal{E}}} \mathit{step}^F(\langle \iota_S, e\rangle, \ell)])[s] \sqsubseteq \mathit{step}^\top(\bot_{\widetilde{S}}, \ell) \sqsubseteq I_\top(\alpha_F(r))[s]$$

This result holds as a special case of the reasoning from the proof of Lemma 3 part 2. □

Comment 8.1 (Soundness of Finitization)**.** Note that in the proofs of Lemmas 3 and 5 we were careful to show that soundness is preserved when any $\widetilde{[\![\mathit{fop}]\!]}$ returns $\top_F$ or if the abstraction process returns $\top_F$, which justifies the soundness of our finitization approach.

B.1.2 Soundness of Combined Interpretation

Proof of Theorem 5. We proceed by cases on the components of $\overline{R}$, for some input argument X.

Case $\vec{\ell} \in \mathcal{L}_A$: We must show that $\alpha_C \circ F(X)[\vec{\ell}] = \alpha_A(F(X))[\vec{\ell}] \sqsubseteq C(\alpha_C(X))[\vec{\ell}] = \widehat{F} \circ \widehat{inj}(\alpha_C(X))[\vec{\ell}] = \widehat{F}(\widehat{inj}(\alpha_C(X)))[\vec{\ell}]$. By assumption, we have that $\alpha_A(F(X))[\vec{\ell}] \sqsubseteq \widehat{F}(\alpha_A(X))[\vec{\ell}]$, so it suffices to show that $\alpha_A(X) \sqsubseteq \widehat{inj}(\alpha_C(X))$. For labels in $\mathcal{L}_A$, the inequality is immediate. For labels in $\mathcal{L}_F$, we have $\alpha_A(X)[\vec{\ell}] \sqsubseteq \widehat{\tau}(\alpha_F(X)[\vec{\ell}]) = \widehat{inj}(\alpha_C(X))[\vec{\ell}]$ from assumption (6.12).

Case $\vec{\ell} \in \mathcal{L}_F$: As above, by the soundness of $I_\top$, we have that $\alpha_F(F(X))[\vec{\ell}] \sqsubseteq I_\top(\alpha_F(X))[\vec{\ell}]$, so it suffices to show that $\alpha_F(X) \sqsubseteq \widetilde{inj}(\alpha_C(X))$. As in the above case, the inequality immediately holds at labels in $\mathcal{L}_F$. Otherwise we have $\forall \vec{\ell}' \in \mathcal{L}_A . \alpha_F(X)[\vec{\ell}'] \sqsubseteq$ $\widetilde{\tau}(\alpha_A(X)[$
$edel'$]) $= \widetilde{inj}(\alpha_C(X))[\vec{\ell}']$ by (6.13).

□

B.1.3 Increased Precision

Lemma 6. $C \circ \widehat{proj} \sqsubseteq_{\widehat{R} \to \overline{R}} \widehat{proj} \circ \widehat{F}$

Proof. Suffices to show that, for an arbitrary $\widehat{r}$, for all $\vec{\ell}$, $C \circ \widehat{proj}(\widehat{r})[\vec{\ell}] \sqsubseteq \widehat{proj} \circ \widehat{F}(\widehat{r})[\vec{\ell}]$. Before proceeding we observe that: $\widetilde{inj} \circ \widehat{proj} = \dot{\widetilde{\tau}}$ and $\widehat{inj} \circ \widehat{proj} = id$.

By cases on whether $\vec{\ell} \in \mathcal{L}_F$ or $\vec{\ell} \in \mathcal{L}_A$:

Case $\vec{\ell} \in \mathcal{L}_A$: Then $C \circ \widehat{proj}(\widehat{r})[\vec{\ell}] = \widehat{F} \circ \widehat{inj} \circ \widehat{proj}(\widehat{r})[\vec{\ell}]$. Because $\widehat{inj} \circ \widehat{proj} = id$, we must show $\widehat{F}(\widehat{r})[\vec{\ell}] \sqsubseteq \widehat{proj} \circ \widehat{F}(\widehat{r})[\vec{\ell}] = \widehat{F}(\widehat{r})[\vec{\ell}]$ which holds trivially as $\widehat{proj}$ is the identity at $\mathcal{L}_A$.

Case $\vec{\ell} \in \mathcal{L}_F$: Then we must show that $I_\top \circ \widetilde{inj} \circ \widehat{proj}(\widehat{r})[\vec{\ell}] = I_\top(\dot{\widetilde{\tau}}(\widehat{r}))[\vec{\ell}] \sqsubseteq \widehat{proj} \circ \widehat{F}(\widehat{r})[\vec{\ell}] = \widetilde{\tau}(\widehat{F}(\widehat{r})[\vec{\ell}]) = \dot{\widetilde{\tau}}(\widehat{F}(\widehat{r}))[\vec{\ell}]$. By assumption, $LB \circ \dot{\widetilde{\tau}} \sqsubseteq \dot{\widetilde{\tau}} \circ \widehat{F}$, so it suffices to show that $I_\top \circ \dot{\widetilde{\tau}}(\widehat{r})[\vec{\ell}] \sqsubseteq LB \circ \dot{\widetilde{\tau}}(r)[\vec{\ell}]$. By cases on the form of $\vec{\ell} \in \mathcal{L}_F$:

Subcase $\vec{\ell} = \ell_f^\circ \rightsquigarrow \ell_f^\bullet$: From the definitions of $I_\top$ and LB, we must show that:

$$step^\top(\widetilde{s}, \ell_f) \sqsubseteq step^{LB}(\widetilde{s}, \ell_f)$$

where $\widetilde{s} = \bigsqcup_{p \in pred(\ell_f)} \dot{\widetilde{\tau}}(\widehat{r})[p^\bullet \rightsquigarrow \ell_f^\circ]$. For branches (6.15) and (6.16) in $step^{LB}$, the corresponding branch (6.5) and (6.6) in $step^\top$ clearly yield identical results. As branch (6.18) returns $\widetilde{s}[x \mapsto \top_F]$, and the corresponding branches (6.7) and (6.8) of $step^\top$ produce states of the form: $\widetilde{s}[x \mapsto v]$ (where v is some member of $\mathbb{P}(V_f)^\top$), the inequality trivially holds. Finally, we do not have to consider branch (6.17) of $step^{LB}$ nor branch (6.9) of $step^F$ because the syntactic constraints of the language rule such cases out for a label of the form ℓ_f.

Finally, if $\ell = s$, we must additionally show that $step^\top(\bot_{\widetilde{S}}, s) \sqsubseteq step^{LB}(\bot_{\widetilde{S}}, s)$. This follows by the same reasoning as above.

Subcase $\vec{\ell} = \ell_f^\bullet \rightsquigarrow \ell^\circ$: By definition, $I_\top(\dot{\widetilde{\tau}}(\widehat{r}))[\ell_f^\bullet \rightsquigarrow \ell^\circ]$ either returns $\dot{\widetilde{\tau}}(\widehat{r})[\ell_f^\circ \rightsquigarrow \ell_f^\bullet]$ or $\bot$. As $LB(\dot{\widetilde{\tau}}(\widehat{r}))[\ell_f^\bullet \rightsquigarrow \ell^\circ] = \dot{\widetilde{\tau}}(\widehat{r})[\ell_f^\circ \rightsquigarrow \ell_f^\bullet]$, the inequality trivially holds.

$\square$

Lemma 7. *Assume for two complete lattices* T *and* $\widehat{\mathsf{T}}$*, a complete-join morphism* $\mu : \widehat{\mathsf{T}} \to \mathsf{T}$*, and two functions* $\mathsf{F} : \mathsf{T} \to \mathsf{T}$*,* $\widehat{\mathsf{F}} : \widehat{\mathsf{T}} \to \widehat{\mathsf{T}}$ *we have that* $\mathsf{F} \circ \mu \sqsubseteq_{\widehat{\mathsf{T}} \to \mathsf{T}} \mu \circ \widehat{\mathsf{F}}$*. Assume then we have two* Ord *termed sequences defined via transfinite recursion as:*

$$\begin{array}{rclcrcl} \mathsf{F}^0 & = & \bot_\mathsf{T} & \quad & \widehat{\mathsf{F}}^0 & = & \bot_{\widehat{\mathsf{T}}} \\ \mathsf{F}^{\delta+1} & = & \mathsf{F}(\mathsf{F}^\delta) & & \widehat{\mathsf{F}}^{\delta+1} & = & \widehat{\mathsf{F}}(\widehat{\mathsf{F}}^\delta) \\ \mathsf{F}^\lambda & = & \bigsqcup_{\beta<\lambda} \mathsf{F}^\beta & & \widehat{\mathsf{F}}^\lambda & = & \bigsqcup_{\beta<\lambda} \widehat{\mathsf{F}}^\beta \end{array}$$

Then for any $\delta \in \mathsf{Ord}$*,* $\mathsf{F}^\delta \sqsubseteq \mu(\widehat{\mathsf{F}}^\delta)$

Proof. By transfinite induction.

Case $\delta = 0$**:** Then $\mathsf{F}^0 = \bot_\mathsf{T} = \mu(\bot_{\widehat{\mathsf{T}}}) = \mu(\widehat{\mathsf{F}}^0)$, where the equality between $\bot$ terms comes from the fact that μ is a complete join-morphism.

Case $\delta + 1$**:** Assume $\mathsf{F}^\delta \sqsubseteq \mu(\widehat{\mathsf{F}}^\delta)$. By the monotonicity of F, we have that $\mathsf{F}^{\delta+1} = \mathsf{F}(\mathsf{F}^\delta) \sqsubseteq \mathsf{F}(\mu(\widehat{\mathsf{F}}^\delta))$, so it suffices to show that $\mathsf{F}(\mu(\widehat{\mathsf{F}}^\delta)) \sqsubseteq \mu(\widehat{\mathsf{F}}^{\delta+1})$. Further, by definition $\widehat{\mathsf{F}}^{\delta+1} = \widehat{\mathsf{F}}(\widehat{\mathsf{F}}^\delta)$, thus we must show that $\mathsf{F} \circ \mu(\widehat{\mathsf{F}}^\delta) \sqsubseteq \mu \circ \widehat{\mathsf{F}}(\widehat{\mathsf{F}}^\delta)$, which holds by assumption.

Case $\delta = \lambda$**:** By the inductive hypothesis, $\forall \beta < \lambda. \mathsf{F}^\beta \sqsubseteq \mu(\widehat{\mathsf{F}}^\beta)$, thus

$$\mathsf{F}^\lambda = \bigsqcup_{\beta<\lambda} \mathsf{F}^\beta \sqsubseteq \bigsqcup_{\beta<\lambda} \mu(\widehat{\mathsf{F}}^\beta) = \mu(\bigsqcup_{\beta<\lambda} \widehat{\mathsf{F}}^\beta) = \mu(\widehat{\mathsf{F}}^\lambda)$$

Where the second-to-last equality follow from μ being a complete join morphism.

□

Proof of Theorem 6. From the definition of *lfp* C and *lfp* $\widehat{\mathsf{F}}$ as the limit of the ordinal termed sequences defined as in Lemma 7, Lemma 6, that $\widehat{\mathsf{proj}}$ is a complete join morphism from part 3 of assumption Equation (6.19), Lemma 7 gives us that *lfp* $\mathsf{C} \sqsubseteq \widehat{\mathsf{proj}}($*lfp* $\widehat{\mathsf{F}})$. As $\widehat{\mathsf{inj}}$ is monotone, we have: $\widehat{\mathsf{inj}}($*lfp* $\mathsf{C}) \sqsubseteq \widehat{\mathsf{inj}} \circ \widehat{\mathsf{proj}}($*lfp* $\widehat{\mathsf{F}})$, whence we have $\widehat{\mathsf{inj}}($*lfp* $\mathsf{C}) \sqsubseteq$ *lfp* $\widehat{\mathsf{F}}$ as $\widehat{\mathsf{inj}} \circ \widehat{\mathsf{proj}} = \mathsf{id}$ as observed in the proof of Lemma 6. □

B.2 *Proofs For Section 6.5*

We note that Equation (B.1) still applies under the updated definition of $\mathcal{V}$.

Updated Proof for Theorem 4. We begin by noting that for the intraprocedural fragment of the language, the proofs of soundness given Appendix B.1.1 generalize naturally to the new definition of states. As an informal argument as to why: note that the definition of $\mathcal{V}$ given above only considers the values of variables in the stack frame of the currently executing method. From the definition of α_F, this in turn implies that the values in the abstracted state are exclusively determined by the concrete values in the active state frame. Finally, as the intraprocedural fragment of the language manipulates only the active stack frame, the arguments made in Appendix B.1.1 translate naturally the extended state definition. Finally, although we have extended the intraprocedural fragment of the language with a `return` statement, the semantics of this statement given by $step^{\top}$ and $step^{F}$ are simply a special case of an assignment statement, and the argument given in the proof of Lemma 3 easily applies to this statement as well.

We therefore only concern ourselves with establishing soundness with respect to the newly defined interprocedural fragment. By Lemma 1, it suffices to establish soundness for each new type of edge individually. Without loss of generality, we proceed to prove soundness for some arbitrary instance of each type of edge and for some r.

Case $\ell_c^{\circ} \rightsquigarrow \ell_c^{\bullet}$: By reasoning similar to Appendix B.1.1, it suffices to show that

$$\alpha_F([\ell_c \mapsto \bigsqcup_{p \in pred(\ell_c)} r[p^{\bullet} \rightsquigarrow \ell_c^{\circ}]])[\ell_c] \sqsubseteq I_{\top}(\alpha_F(r))[\ell_c]$$

$$= \bigsqcup_{p \in pred(\ell_c)} \alpha_F(r)[p^{\bullet} \rightsquigarrow \ell_c^{\circ}]$$

From the definition of the domain R, we have:

$$\alpha_F([\ell_c \mapsto \bigsqcup_{p \in \mathtt{pred}(\ell_c)} r[p^\bullet \rightsquigarrow \ell_c^\circ]]) = \alpha_F(\bigsqcup_{\substack{p \in \mathtt{pred}(\ell_c) \\ \mathtt{in} \in r[p^\bullet \rightsquigarrow \ell_c^\circ]}} [\ell_c \mapsto \{\mathtt{in}\}])$$

$$= \bigsqcup_{\substack{p \in \mathtt{pred}(\ell_c) \\ \mathtt{in} \in r[p^\bullet \rightsquigarrow \ell_c^\circ]}} \alpha_F([\ell_c \mapsto \{\mathtt{in}\}])$$

By the definition of least upper bounds, it suffices to show that, for some arbitrary $p' \in \mathtt{pred}(\ell_c), \mathtt{in} \in r[p'^\bullet \rightsquigarrow \ell_c^\circ]$:

$$\alpha_F([\ell_c \mapsto \{\mathtt{in}\}])[\ell_c] \sqsubseteq \bigsqcup_{p \in \mathtt{pred}(\ell_c)} \alpha_F(r)[p^\bullet \rightsquigarrow \ell_c^\circ]$$

We show that $\alpha_F([\ell_c \mapsto \{\mathtt{in}\}])[\ell_c] \sqsubseteq \alpha_F(r)[p'^\bullet \rightsquigarrow \ell_c^\circ]$, whence by the definition of least upper bounds and transitivity we will have the desired result.

For some arbitrary variable v, from the fact that $\mathtt{in} \in r[p'^\bullet \rightsquigarrow \ell_c^\circ]$, it is immediate that:

$$\mathcal{V}([\ell_c \mapsto \{\mathtt{in}\}][\ell_c], v) = \mathcal{V}(\{\mathtt{in}\}, v) \subseteq \mathcal{V}(r[p'^\bullet \rightsquigarrow \ell_c^\circ], v)$$

whence by Equation (B.1) we have that $\alpha_F([\ell_c \mapsto \{\mathtt{in}\}])[\ell_c][v] \sqsubseteq \alpha_F(r)[p'^\bullet \rightsquigarrow \ell_c^\circ][v]$, as required.

Case $p^\bullet \rightsquigarrow \ell_r^\circ$: Trivial, by the definition of α_F and reasoning similar to the above.

Case $\ell_c^\bullet \rightsquigarrow \ell'^\circ$: We first abbreviate $\ell_c^\bullet \rightsquigarrow \ell'^\circ$ as $\vec{c}$ and (as usual) $\ell_c^\circ \rightsquigarrow \ell_c^\bullet$ as ℓ_c. By reasoning similar to that in case $\ell_c^\circ \rightsquigarrow \ell_c^\bullet$, it suffices to show that:

$$\bigsqcup_{\langle s \circ s_r, \mathcal{R}, E\rangle \in r[\ell_c]} \alpha_F([\vec{c} \mapsto \{\langle \vec{s} \circ s_r \circ [p \mapsto s_r[y]], \mathcal{R} \circ \ell, E\rangle\}])[\vec{c}] \sqsubseteq [p \mapsto \alpha_F(r)[\ell_c][y]]$$

for which it suffies to show, for some arbitrary $\langle \vec{s} \circ s_r, \mathcal{R}, E\rangle \in r[\ell_c]$ that:

$$\alpha_F([\vec{c} \mapsto \mathcal{K}])[\vec{c}] \sqsubseteq [p \mapsto \alpha_F(r)[\ell_c][y]]$$

where $\mathcal{K} = \{\langle \vec{s} \circ s_r \circ [p \mapsto s_r[y]], \mathcal{R} \circ \ell, E\rangle\}$. From the new definition on $\mathcal{V}$, for any variable $v \neq p$, it is immediate that $\mathcal{V}(\mathcal{K}, v) = \emptyset$, whence we have that:

$$\begin{aligned}\alpha_F([\vec{c} \mapsto \mathcal{K}])[\vec{c}][v] &= \alpha_F(\bot_R)[\vec{c}][v] \\ &= \bot_{\widetilde{R}}[\vec{c}][v] = \big[p \mapsto \alpha_F(r)[\ell_c][y]\big][v]\end{aligned}$$

It therefore remains to show that

$$\alpha_F([\vec{c} \mapsto \mathcal{K}])[\vec{c}][p] \sqsubseteq \big[p \mapsto \alpha_F(r)[\ell_c][y]\big][p] = \alpha_F(r)[\ell_c][y]$$

From the definition of $\mathcal{K}$, we have that

$$\mathcal{V}(\mathcal{K}, p) = \{s_r[y]\} \subseteq \mathcal{V}(r[\ell_c], y)$$

whence by Equation (B.1) we have the desired result.

Case $\ell_r^{\circ} \rightsquigarrow \ell_r^{\bullet}$: We will prove that, for some some arbitrary r, $p \in \mathtt{pred}(\ell_r)$, $\langle \vec{s} \circ s_r \circ s_c, \mathcal{R}, E\rangle \in r[p^{\bullet} \rightsquigarrow \ell_r^{\circ}]$ where $\langle \vec{s} \circ s_r, \mathcal{R}, E\rangle \in r[\ell_c^{\circ} \rightsquigarrow \ell_c^{\bullet}]$ that:

$$\alpha_F([\ell_r \mapsto \mathcal{J}])[\ell_r] \sqsubseteq \alpha_F(r)[\ell_c]\big[x \mapsto \alpha_F(r)[p^{\bullet} \rightsquigarrow \ell_r^{\circ}][\rho]\big]$$

where $\mathcal{J} = \{\langle \vec{s} \circ s_r[x \mapsto s_c[\rho]], \mathcal{R}, E\rangle\}$

For any variable $v \neq x$, we have that $\mathcal{V}(\mathcal{J}, v) \subseteq \mathcal{V}(r[\ell_c], v)$, whence the inequality holds by Equation (B.1). For the variable x, we have that $\mathcal{V}(\mathcal{J}, x) \subseteq \mathcal{V}(r[p^{\bullet} \rightsquigarrow \ell_r^{\circ}], \rho)$, whence the inequality again holds by assumption on α_F.

□

B.3 Formalisms and Proofs for Subfixpoint Iteration

Before defining subfixpoint iteration, we introduce the following notation. Let $m : T \to U$ be a map, where U is a complete lattice. For $S \subseteq T$, define:

$$m\|_S = \lambda x. \begin{cases} m[x] & x \in S \\ \bot_U & o.w. \end{cases}$$

We will abuse notation and for an element $\langle \widehat{m}, \widetilde{m} \rangle : \overline{R}$, define $\langle \widehat{m}, \widetilde{m} \rangle|_{\mathcal{L}_A} = \widehat{m}$ and $\langle \widehat{m}, \widetilde{m} \rangle|_{\mathcal{L}_F} = \widetilde{m}$.

We define the subfixpoint iteration process as a function $M : \overline{R} \to \overline{R}$:

$$M(X) = \Big\langle \bigsqcup_{i<\omega} U^i(\widehat{inj}(X))|_{\mathcal{L}_A}, \bigsqcup_{i<\omega} T(\widetilde{inj}(X)\|_{\mathcal{L}_A})^i(\bot)|_{\mathcal{L}_F} \Big\rangle \qquad U(X) = X\|_{\mathcal{L}_F} \sqcup \widehat{F}(X)\|_{\mathcal{L}_A}$$

$$T(m)(X) = m\|_{\mathcal{L}_A} \sqcup I_\top(X)\|_{\mathcal{L}_F}$$

In the above definitions, the U and T play the role of iterating the two interpreters to fixpoint.[1] In the definition of U, after abstract semantic function $\widehat{F}$ is applied to the input argument, the results from this application in $\mathcal{L}_F$ are discarded in favor of the values in the input argument X. Thus, while iterating $U^i(\widehat{inj}(X))$, the states at flow edges $\mathcal{L}_F$ are effectively fixed to $\widehat{inj}(X)\|_{\mathcal{L}_F}$, while the states at flow edges $\mathcal{L}_A$ evolve through repeated application of $\widehat{F}$. The definition of T is similar; except the state information in application code is fixed while the state information in the framework evolves via application of $I_\top$. Unlike the $\sqcup_{i<\omega} U^i(\widehat{inj}(X))$ term, $\sqcup_{i<\omega} T(\widetilde{inj}(X)\|_{\mathcal{L}_A})^i(\bot)$ iterates from bottom, discarding information computed in previous rounds of iteration as described in Section 6.6.

We first prove that M is monotone.

Lemma 8.

1. $\forall m, m', X, X' : \widetilde{R}. m \sqsubseteq m' \land X \sqsubseteq X' \Rightarrow T(m)(X) \sqsubseteq T(m')(X')$
2. $\forall m, m', X, X' : \widetilde{R}. m \sqsubseteq m' \land X \sqsubseteq X' \Rightarrow \sqcup_{i<\omega} T(m)^i(X) \sqsubseteq \sqcup_{i<\omega} T(m')^i(X')$
3. $\forall X, X' : \widehat{R}. X \sqsubseteq X' \Rightarrow U(X) \sqsubseteq U(X')$
4. $\forall X, X' : \widehat{R}. X \sqsubseteq X' \Rightarrow \sqcup_{i<\omega} U^i(X) \sqsubseteq \sqcup_{i<\omega} U^i(X')$

[1] We have not justified here that the term $\sqcup_{i<\omega} U^i(\widehat{inj}(X))$ is a fixpoint of U, nor that a fixed point of this function exists. However, we chose this definition for the parallel to the widening definitions presented below; which we prove converges to a fixed-point. If $\widehat{F}$ is ω-upper-continuous, then the U term converges to a fixpoint, as proved in Appendix B.3.5.

Proof.

1. By cases on whether $\vec{\ell} \in \mathcal{L}_A$ or $\vec{\ell} \in \mathcal{L}_F$:

   Case $\vec{\ell} \in \mathcal{L}_F$: $T(m)(X)[\vec{\ell}] = I_T(X)[\vec{\ell}] \sqsubseteq I_T(X')[\vec{\ell}] = T(m')(X')[\vec{\ell}]$ by the monotonicity of I_T (by Lemma 18, proved below).

   Case $\vec{\ell} \in \mathcal{L}_A$: $T(m)(X)[\vec{\ell}] = m[\vec{\ell}] \sqsubseteq m'[\vec{\ell}] = T(m')(X')[\vec{\ell}]$ by assumption.

2. By induction, using the monotonicity of T shown above it can be shown that $\forall i < \omega . T(m)^i(X) \sqsubseteq T(m')^i(X')$, whence the result follows from the definition of least upper bounds.

3. Immediate from the fact that $\widehat{F}$ is monotone.

4. By similar proof to (2) above.

□

Theorem 9. *M is monotone.*

Proof. Immediate from parts 2 and 4 of Lemma 8 and the monotonicity of $\tilde{\tau}$ and $\hat{\tau}$. □

We next define two ordinal termed sequences $\mathfrak{C}^{\delta \in \mu}$ and $\mathfrak{M}^{\delta \in \mu}$, where μ is an ordinal defined as in [48], via transfinite recursion as:

$$\begin{array}{rclrcl} \mathfrak{C}^0 & = & \bot_{\overline{R}} & \mathfrak{M}^0 & = & \bot_{\overline{R}} \\ \mathfrak{C}^{\delta+1} & = & C(\mathfrak{C}^\delta) & \mathfrak{M}^{\delta+1} & = & M(\mathfrak{M}^\delta) \\ \mathfrak{C}^\lambda & = & \bigsqcup_{\beta<\lambda} \mathfrak{C}^\beta & \mathfrak{M}^\lambda & = & \bigsqcup_{\beta<\lambda} \mathfrak{M}^\beta \end{array}$$

As both M and C are monotone functions over complete lattices, then by Corollary 3.3 of [48], the limit of $\mathfrak{C}^{\delta \in \mu}$, $\mathfrak{C}^\epsilon$, exists and is the least fixpoint of C and the limit of $\mathfrak{M}^{\delta \in \mu}$ also exists and is a least fixed point of M.

Before proceeding, we first show how to (almost) instantiate the asynchronous chaotic iteration with memory strategy defined by [46] to give an equivalent definition of subfixpoint iteration. (In the following, for continuity of notation, we will use F as the monotone function over a lattice $(L^n)^m \to L^n$, and *not* the concrete semantic function of Section 6.3.2.)

Assume some isomorphism ξ between $\{0, \ldots, |\mathcal{L}|\}$ and $\mathcal{L}$. We can define an indexed representation of an element of $\overline{R}$, where X_i corresponds to the mostly-concrete or abstract state at $\xi(i)$. We take $m = 3$, and for any i such that $\xi(i) \in \mathcal{L}_A$, define:

$$F_i(I, E, D) = \widehat{F}(\widehat{inj}(\eta(I, E)))[\xi(i)]$$
$$\eta(I, E) = \left\langle \lambda\vec{\ell}_a : \mathcal{L}_A.I_{\xi^{-1}(\vec{\ell}_a)},\ \lambda\vec{\ell}_f : \mathcal{L}_F : E_{\xi^{-1}(\vec{\ell}_f)} \right\rangle$$

η transforms the indexed representation into the product-of-maps representation expected by $\widehat{inj}$. When constructing the abstract state map it uses the values from I, and those from E to construct the mostly-concrete map.

For i such that $\xi(i) \in \mathcal{L}_F$, define:

$$F_i(I, E, D) = I_{\top}(\widetilde{inj}(\chi(D)))[\xi(i)]$$
$$\chi(D) = \left\langle \lambda\vec{\ell}_a : \mathcal{L}_A.D_{\xi^{-1}(\vec{\ell}_a)},\ \lambda\vec{\ell}_f : \mathcal{L}_F.D_{\xi^{-1}(\vec{\ell}_f)} \right\rangle$$

Like η, χ transforms the indexed representation into a map representation, using D for states in $\mathcal{L}_F$ and $\mathcal{L}_A$. Clearly, $F \circ \sigma = C$.

We will now define the ordinal termed sequences S of indices that specify the values of I, E, and D. When indexing S, we take $I = 1$, $E = 2$, and $D = 3$. Then:

$$(S_I^{\delta+1})_i = \delta$$
$$(S_E^{\delta+1})_i = \lambda \text{ (where } \lambda \text{ is the largest limit ordinal} \leqslant \delta)$$
$$(S_D^{\delta+1})_i = \begin{cases} S_E^{\delta+1} & \xi(i) \in \mathcal{L}_A \\ 0 & \xi(i) \in \mathcal{L}_F \wedge \delta = \lambda \text{ (where } \lambda \text{ is a limit ordinal)} \\ \delta & o.w. \end{cases}$$

A value of δ' in $(S_Y^{\delta+1})_i$ indicates that index i of sequence $Y(\in \{I, E, D\})$ should hold the value of the δ'^{th} iterate of F_i. We do not define the value of S_j at the limit ordinals as they are not used during iteration, we assume they take some value consistent with the requirements given in [46].

These definitions imply that I always holds the values computed in the last round of iteration, E holds the values computed at the most recent limit ordinal, and D holds the values of at the most recent limit ordinal for states at $\mathcal{L}_A$, $\bot_{\widetilde{S}}$ for $\mathcal{L}_F$ at the immediate successor to some limit ordinal, and the value from the previous iteration otherwise (we assume that $X_i^0 = \bot$ for all i).

It should be clear that the value computed at $M(\bot_{\overline{R}})[\vec{\ell}]$ corresponds to the values computed at $X_{\xi^{-1}(\vec{\ell})}$ at ω. However, our limiting behavior at all other limit ordinals is different: M takes the limits of new sequences, whereas at the limit ordinals the sequence defined in [46] take the limit over the entire preceding sequence of values. In addition, the definition of S_D violates requirement 3.1.(d). As a result, although we have that the limit of $\mathfrak{M}^\delta$ exists and is a least fixed point of M, it is not immediate that this is equal to *lfp* C.

We therefore directly prove our desired result:

Theorem 10. *lfp* $M =$ *lfp* C

B.3.1 Technical Lemmas

Before proving Theorem 10 we need the following technical lemmas.

Lemma 9. *The function* $T(m)(X)$ *is* ω*-upper-continuous in its second argument.*

Proof.

$$
\begin{aligned}
\bigsqcup_n T(m)(x_n) &= \bigsqcup_n I_\top(x_n)\|_{\mathcal{L}_F} \sqcup m\|_{\mathcal{L}_A} \\
&= m\|_{\mathcal{L}_A} \sqcup \bigsqcup_n I_\top(x_n)\|_{\mathcal{L}_F} \\
&= m\|_{\mathcal{L}_A} \sqcup I_\top(\bigsqcup_n x_n)\|_{\mathcal{L}_F} \text{ (by the } \omega\text{-upper-continuity of } I_\top\text{, see Appendix B.3.1)} \\
&= T(m)(\bigsqcup_n x_n)
\end{aligned}
$$

□

Theorem 11. *The limit of* $T(m)^{(i \in \mathbb{N})}(\bot)$ *exists and is a fixpoint of* $T(m)$ *for any* m.

Proof. Follows immediately from Lemma 9 and Kleene's fixpoint theorem. □

Let us recall the following facts from [48]:

1. The least upper bound of a (potentially infinite) set S of post-fixed points of a monotone function F is itself a post-fixed point of F

2. For a monotone function $g : L \to L$ on a complete lattice L, the ordinal termed sequence defined as:

$$
\begin{aligned}
\mathfrak{G}^0 &= \bot_L \\
\mathfrak{G}^{\delta+1} &= g(\mathfrak{G}^\delta) \\
\mathfrak{G}^\lambda &= \bigsqcup_{\alpha<\lambda} \mathfrak{G}^\alpha
\end{aligned}
$$

   is increasing

Next define the family of sequences $\mathfrak{T}_m^\delta, \delta \in \mathrm{Ord}$ via transfinite recursion as:

$$
\begin{aligned}
\mathfrak{T}_m^0 &= \bot_{\tilde{R}} \\
\mathfrak{T}_m^{\delta+1} &= T(m)(\mathfrak{T}_m^\delta) \\
\mathfrak{T}_m^\lambda &= \bigsqcup_{\alpha<\lambda} \mathfrak{T}_m^\alpha
\end{aligned}
$$

By the monotonicity of T we have that all such sequences are increasing, and by Theorem 11 the limit of all sequences exist and is $\mathfrak{T}_m^\omega$.

Lemma 10. $\forall m, \delta \in \mathrm{Ord}.\mathfrak{T}_m^\delta \sqsubseteq \mathfrak{T}_m^\omega$

Proof. Immediate from the definition of $\mathfrak{T}_m^\delta$ and by the fact that $\mathfrak{T}_m^\omega$ is the least fixpoint of $T(m)$. □

Continuity of $I_\top$

We now demonstrate the ω-upper-continuity of $I_\top$ on increasing chains of $\widetilde{R}$.

Lemma 11. *For any increasing chain* $x_i : \mathbb{P}(V_f)^\top$, $v \in \bigsqcup_i x_i \Rightarrow \exists k.v \in x_k$

Proof. First, observe that by the definition of least upper bound, if $(\bigsqcup_i x_i) \neq \top_F$, then $\forall i, x_i, \neq \top_F$. Thus, by the definition of $\bigsqcup$ in $\mathbb{P}(V_f)^\top$ as a union over the sets x_i, $v \in \bigsqcup_i x_i$ implies there is some set in the chain x_i, that contains v. We let j be the smallest such x_j containing x_j. □

Corollary 11.1. *For an increasing chain of products* $c_i \equiv \langle x_1, \ldots, x_n \rangle_i : \mathbb{P}(V_f)^\top \times \ldots \mathbb{P}(V_f)^\top$, *if there exists some* $\langle v_1, \ldots, v_n \rangle$ *such that* $\forall j.v_j \in \pi_j(\bigsqcup_i c_i)$, *then there exists some* k *such that* $\forall j.v_j \in \pi_j(c_k)$.

Proof. By pointwise application of Lemma 11, for each j there must be some k_j such that $v_j \in \pi_j(c_k)$. Let m be the max over these k_j. As the chain c_i is increasing, if $v_j \in \pi_j(c_{k_j})$, then necessarily $v_j \in \pi_j(c_m)$. □

Lemma 12. $\widetilde{[\![fop]\!]}$ *is* ω*-upper-continuous for all increasing chains* $arg_i : \mathbb{P}(V_f)^\top \times \ldots \times \mathbb{P}(V_f)^\top$

Proof. Consider the case where, for some j and k, $\pi_j(arg_k) = \top_F$. Then $\widetilde{[\![fop]\!]}(arg_k) = \top_F$, and further $\pi_j(\bigsqcup_i arg_i) = \top_F$ whence $\widetilde{[\![fop]\!]}(\bigsqcup_i arg_i) = \top_F$.

Consider now the case where the chain contains no $\top_F$ values. Then, by the definition of $[\![fop]\!]$ and $\widetilde{[\![fop]\!]}$, the sequence $\widetilde{[\![fop]\!]}(arg_i)$ will not contain $\top_F$, and thus $\bigsqcup_i \widetilde{[\![fop]\!]}(arg_i)$ will not equal $\top_F$, and similarly $\widetilde{[\![fop]\!]}(\bigsqcup_i arg_i) \neq \top_F$.

It therefore suffices to show that $v \in \bigsqcup_i \widetilde{[\![fop]\!]}(arg_i) \Leftrightarrow v \in \widetilde{[\![fop]\!]}(\bigsqcup_i arg_i)$.

Case $\Rightarrow$: By the definition of lub and $\widetilde{[\![fop]\!]}$, $\exists k.v \in [\![fop]\!](arg_k)$. But then $\forall j.\pi_j(arg_k) \sqsubseteq \pi_j(\bigsqcup_i arg_i)$ and thus by definition of $\widetilde{[\![fop]\!]}$, $v \in \widetilde{[\![fop]\!]}(\bigsqcup_i arg_i)$.

Case $\Leftarrow$: If $v \in \widetilde{[\![fop]\!]}(\bigsqcup_i arg_i)$, then there must exist some set of values v_j and some $E \in \mathcal{E}$ such that $\forall j.v_j \in \pi_j(\bigsqcup_i arg_i)$ and $[\![fop]\!](E, v_0, \ldots, v_n) = \langle v, _\rangle$. By Corollary 11.1, there must be some k, such that $\forall j.v_j \in \pi_j(arg_k)$, whence $v \in \widetilde{[\![fop]\!]}(arg_k) \sqsubseteq \bigsqcup_i \widetilde{[\![fop]\!]}(arg_i)$.

□

Lemma 13. *$\widetilde{[\![fop]\!]}$ is pointwise monotone.*

Proof. Consider the case of some $\widetilde{[\![fop]\!]}(v_0, \ldots, v_n)$ where one of $v_i = \top_F$. Then, $\widetilde{[\![fop]\!]}(v_0, \ldots, v_n) = \top_F$, and for any v'_j, $v_j \sqsubseteq v'_j \Rightarrow v'_j = \top_F$, whence $\widetilde{[\![fop]\!]}(v'_0, \ldots, v'_n) = \top_F$, for any values v'_i pointwise greater than v_i. Next, consider the case of some arguments $v_i \sqsubseteq v'_i$, where $\forall j.v_j \neq \top_F$. Then $\widetilde{[\![fop]\!]}(v_0, \ldots, v_n) \neq \top_F$. If any of v'_i is $\top_F$, then monotonicity holds trivially. Finally, consider when, for each j, $v_j \subseteq v'_j$. Then by the definition of $\widetilde{[\![fop]\!]}$, it is immediate that $\widetilde{[\![fop]\!]}(v_0, \ldots, v_n) \subseteq \widetilde{[\![fop]\!]}(v'_0, \ldots, v'_n)$. □

Lemma 14. *If $s_i : \widetilde{S}$ is an increasing sequence, and if for such sequences $\bigsqcup_i f(s_i) = f(\bigsqcup_i s_i)$, then: $\forall x.Q = \bigsqcup_i s_i[x \mapsto f(s_i)] = (\bigsqcup s_i)[x \mapsto f(\bigsqcup_i s_i)] = R$.*

Proof. $Q[x] = \bigsqcup_i f(s_i) = f(\bigsqcup_i s_i) = R[x]$ and $Q[y \neq x] = \bigsqcup_i s_i[y] = (\bigsqcup_i s_i)[y] = R[y]$. □

Lemma 15. *For an increasing sequence $s_i : \widetilde{S}$, $\forall \ell. \bigsqcup_i step^\top(s_i, \ell) = step^\top(\bigsqcup_i s_i, \ell)$.*

Proof. By cases on the branch taken in *step*$^\top$:

Branch (6.5): Trivial

Branches (6.6, 6.7, 6.9): Follows from the Lemma 14 and that $\lambda_.\top_{\widehat{A}}$, $\lambda_.\{[\![bc_f]\!]\}$, and $\lambda s.s[y]$ are trivially continuous on increasing sequences.

Branch (6.8): From Lemma 14 and Lemma 12.

□

Lemma 16. *$step^{\top}$ is monotone.*

Proof. Immediate from the monotonicity of $\widetilde{[\![fop]\!]}$ (Lemma 13), and the definition of $step^{\top}$.

□

Lemma 17.

1. $\forall \tilde{r}, \tilde{r}', p^{\bullet} \rightsquigarrow \ell^{\circ}. \widetilde{\mathcal{FT}}(\tilde{r}, p^{\bullet} \rightsquigarrow \ell^{\circ}) \wedge \tilde{r} \sqsubseteq \tilde{r}' \Rightarrow \widetilde{\mathcal{FT}}(\tilde{r}', p^{\bullet} \rightsquigarrow \ell^{\circ})$

2. *For an increasing sequence r_i, $\forall p^{\bullet} \rightsquigarrow \ell^{\circ}. \widetilde{\mathcal{FT}}(\bigsqcup_i r_i, p^{\bullet} \rightsquigarrow \ell^{\circ}) \Leftrightarrow \exists j. \widetilde{\mathcal{FT}}(r_j, p^{\bullet} \rightsquigarrow \ell^{\circ})$*

Proof.

1. Let $\tilde{s} = \tilde{r}[p]$ and $\tilde{s}' = \tilde{r}'[p]$. Consider by cases why $\widetilde{\mathcal{FT}}$ returned true:

   Branch 6.2 Then $\tilde{s}[x]\widetilde{[\![\texttt{<=>}]\!]}\tilde{s}[y]$, thus either $\tilde{s}[x] = \top_F$, $\tilde{s}[y] = \top_F$, or $v \in \tilde{s}[x]$ and $v' \in \tilde{s}[y]$ such that $v[\![\texttt{<=>}]\!]v'$. In the former two cases, $\tilde{s}'[x]$ or $\tilde{s}'[y]$ must also $\top_F$, and thus $\tilde{s}'[x]\widetilde{[\![\texttt{<=>}]\!]}\tilde{s}'[y]$. Consider the final case. Then by definition of $\sqsubseteq$, $\tilde{s}'[x] = \top_F$ or $\tilde{s}[x] \subseteq \tilde{s}'[x]$ and similarly for $\tilde{s}[y]$ and $\tilde{s}'[y]$. In the case where either $\tilde{s}'[x]$ or $\tilde{s}'[y]$ is $\top_F$, then $\tilde{s}'[x]\widetilde{[\![\texttt{<=>}]\!]}\tilde{s}'[y]$ trivially returns true. Otherwise, we have $v \in \tilde{s}'[x]$ and similarly for v' and $\tilde{s}'[y]$, and thus by definition $\tilde{s}'[x]\widetilde{[\![\texttt{<=>}]\!]}\tilde{s}'[y]$ holds.

   Branch 6.3 By similar reasoning to the above, but on the definition of $\widetilde{[\![\texttt{<}\neq\texttt{>}]\!]}$.

   Branch 6.4 $\widetilde{\mathcal{FT}}$ must always return true.

2. For the forward impliciation, consider by cases why $\widetilde{\mathcal{FT}}$ returned true:

   Branch 6.2 By similar reasoning to the above, and without loss of generality, let us consider the case where $(\bigsqcup_i r_i)[p][x] = \top_F$. Then, there must exist some j such that $r_j[p][x] = \top_F$, and thus $\widetilde{\mathcal{FT}}(r_j, p^{\bullet} \rightsquigarrow \ell^{\circ})$ trivially returns true. The

case when $(\bigsqcup_i r_i)[p][y] = \top_F$ follows from similar reasoning. Finally, let us consider the case where $v \in (\bigsqcup_i r_i)[p][x]$, $v' \in (\bigsqcup_i r_i)[p][y]$, and $v[\![<=>]\!]v'$. By Corollary 11.1, there must exists some j, such that $v \in r_j[p][x]$ and $v' \in r_j[p][y]$, whence $r_j[p][x]\widetilde{[\![<=>]\!]}r_j[p][y]$ and thus $\widetilde{\mathcal{FT}}(r_j, p^\bullet \rightsquigarrow \ell^\circ)$ must return true.

Branch 6.3 By similar reasoning to the above.

Branch 6.4 $\forall j.\widetilde{\mathcal{FT}}(r_j, p, \ell)$ must be true.

The backward impliciation follows immediately from part one of the lemma and from the fact that $\forall j.r_j \sqsubseteq \bigsqcup_i r_i$.

□

Lemma 18. $I_\top$ *is monotone*

Proof. Consider some $\widetilde{r} \sqsubseteq \widetilde{r}'$, and some arbitrary $\ell^\circ \rightsquigarrow \ell^\bullet$:

$$\begin{aligned} I_\top(\widetilde{r})[\ell^\circ \rightsquigarrow \ell^\bullet] &= \mathit{step}^\top(\bigsqcup_{p \in \mathrm{pred}(\ell)} \widetilde{r}[p^\bullet \rightsquigarrow \ell^\circ], \ell) \\ &\sqsubseteq \mathit{step}^\top(\bigsqcup_{p \in \mathrm{pred}(\ell)} \widetilde{r}'[p^\bullet \rightsquigarrow \ell^\circ], \ell) \text{ (Lemma 16)} \\ &= I_\top(\widetilde{r}')[\ell] \end{aligned}$$

We have ignored the initialization terms, as it is constant and by the monotonicity of least upper bounds (as $\mathit{step}^\top$ is monotone) monotonicity is preserved.

Next, consider some $p^\bullet \rightsquigarrow \ell^\circ$, and whether $\widetilde{\mathcal{FT}}(\widetilde{r}, p^\bullet \rightsquigarrow \ell^\circ)$ is true. If so, then Lemma 17 implies that $\widetilde{\mathcal{FT}}(\widetilde{r}', p^\bullet \rightsquigarrow \ell^\circ)$ must also be true, which from the fact that $\widetilde{r}[p^\circ \rightsquigarrow p^\bullet] \sqsubseteq \widetilde{r}'[p^\circ \rightsquigarrow p^\bullet]$, gives us that $I_\top(\widetilde{r})[p^\bullet \rightsquigarrow \ell^\circ] \sqsubseteq I_\top(\widetilde{r}')[p^\bullet \rightsquigarrow \ell^\circ]$. Otherwise, $I_\top(\widetilde{r})[p^\bullet \rightsquigarrow \ell^\circ] = \bot \sqsubseteq I_\top(\widetilde{r}')[p^\bullet \rightsquigarrow \ell^\circ]$. □

We can now prove the continuity of $I_\top$.

Theorem 12. *For an increasing sequence* $r_i : \widetilde{R}$, $I_\top$ *is continuous.*

Proof. We first note that the initialization term of $I_\top$ at the start label s is a constant term and can be easily factored out.

It suffices to consider an arbitrary $\ell^\circ \rightsquigarrow \ell^\bullet$ and $p^\bullet \rightsquigarrow \ell^\circ$.

$$\begin{aligned}
\bigsqcup_i I_\top(r_i)[\ell^\circ \rightsquigarrow \ell^\bullet] &= \bigsqcup_i \mathit{step}^\top(\bigsqcup_{p \in \mathit{pred}(\ell)} r_i[p^\bullet \rightsquigarrow \ell^\circ], \ell) \\
&= \mathit{step}^\top(\bigsqcup_i \bigsqcup_{p \in \mathit{pred}(\ell)} r_i[p^\bullet \rightsquigarrow \ell^\circ], \ell) \\
&= \mathit{step}^\top(\bigsqcup_{p \in \mathit{pred}(\ell)} (\bigsqcup_i r_i)[p^\bullet \rightsquigarrow \ell^\circ], \ell) \\
&= I_\top(\bigsqcup_i r_i)[\ell]
\end{aligned}$$

The second equality follows from $\sqcup_{p \in pred(\ell)} r_i[p^\bullet \rightsquigarrow \ell^\circ]$ being an increasing sequence and Lemma 15.

Next, let us consider some $p^\bullet \rightsquigarrow \ell^\circ$. Consider the case where $\widetilde{\mathcal{FT}}(\bigsqcup_i r_i, p^\bullet \rightsquigarrow \ell^\circ)$ is false, thus $I_\top(\bigsqcup_i r_i)[p^\bullet \rightsquigarrow \ell^\circ] = \bot$. By Lemma 17 (2), we have that $\forall j. \widetilde{\mathcal{FT}}(r_j, p^\bullet \rightsquigarrow \ell^\circ)$ is also false, and thus $\bigsqcup_i I_\top(r_i)[p^\bullet \rightsquigarrow \ell^\circ] = \bigsqcup_i \bot = I_\top(\bigsqcup_i r_i)[p^\bullet \rightsquigarrow \ell^\circ]$.

Finally, consider the case where $\widetilde{\mathcal{FT}}(\bigsqcup_i r_i, p^\bullet \rightsquigarrow \ell^\circ)$ is true, whence $I_\top(\bigsqcup_i r_i,)[p^\bullet \rightsquigarrow \ell^\circ] = \bigsqcup_i r_i[p^\circ \rightsquigarrow p^\bullet]$. By Lemma 17, there exists some j such that $\forall k \geqslant j. \widetilde{\mathcal{FT}}(r_k, p^\bullet \rightsquigarrow \ell^\circ)$ is true. Note that Lemma 17 part 2 implies there exists some j such that $\widetilde{\mathcal{FT}}(r_j, \pi)$, and as the sequence r_i is increasing the monotonicity of $\widetilde{\mathcal{FT}}$ (Lemma 17 part 1) implies that $\widetilde{\mathcal{FT}}$ will be true for all indices after j. As $I_\top$ is monotone, we have that $I_\top(r_i)$ is an increasing sequence, and thus

$$\begin{aligned}
[\bigsqcup_i I_\top(r_i)][p^\bullet \rightsquigarrow \ell^\circ] &= [\bigsqcup_{k \geqslant j} I_\top(r_k)][p^\bullet \rightsquigarrow \ell^\circ] \text{ } (I_\top(r_i) \text{ is increasing}) \\
&= \bigsqcup_{k \geqslant j} r_k[p^\circ \rightsquigarrow p^\bullet] \text{ (As } \forall k \geqslant j. \widetilde{\mathcal{FT}}(r_k, p^\bullet \rightsquigarrow \ell^\circ) \text{ must be true)} \\
&= \bigsqcup_i r_i[p^\circ \rightsquigarrow p^\bullet] \text{ } (r_i \text{ is increasing}) \\
&= I_\top(\bigsqcup_i r_i)[p^\bullet \rightsquigarrow \ell^\circ]
\end{aligned}$$

∎

B.3.2 $\mathfrak{C}^\epsilon \sqsubseteq \mathfrak{M}^\epsilon$

We first need the following lemmas.

Lemma 19. $\forall \vec{\ell} \in \mathcal{L}_A, X \sqsubseteq X' \in \overline{R}.\widehat{F} \circ \widehat{inj}(X)[\vec{\ell}] \sqsubseteq \bigsqcup_{i<\omega} U^i(\widehat{inj}(X'))[\vec{\ell}]$

Proof.

$$\widehat{F}(\widehat{inj}(X))[\vec{\ell}] \sqsubseteq \widehat{F}(\widehat{inj}(X'))[\vec{\ell}] = U(\widehat{inj}(X')[\vec{\ell}] \sqsubseteq \bigsqcup_{i<\omega} U^i(\widehat{inj}(X'))[\vec{\ell}]$$

∎

Lemma 20. $\forall m, \ell \in \mathcal{L}_A, \delta \in \mathrm{Ord}.\delta > 0 \Rightarrow \mathfrak{T}_m^\delta[\ell] = m[\ell]$

Proof. By an easy transfinite induction argument, at a successor ordinal $\delta+1$: $\mathfrak{T}_m^{\delta+1}[\ell] = T(m)(\mathfrak{T}_m^\delta)[\ell] = m[\ell]$

At a limit ordinal, we have that $\mathfrak{T}_m^\lambda[\ell] = \bigsqcup_{\alpha<\lambda} \mathfrak{T}_m^\alpha[\ell] = \bot \sqcup \bigsqcup_{\alpha<\lambda} \mathfrak{T}_m^{\alpha+1}[\ell] = \bigsqcup_{\alpha<\lambda} \mathfrak{T}_m^{\alpha+1}[\ell] = \bigsqcup_{\alpha<\lambda} m[\ell] = m[\ell]$ ∎

Lemma 21. $\forall \delta \in \mu, m.\mathfrak{C}^\delta|_{\mathcal{L}_A} \sqsubseteq m|_{\mathcal{L}_A} \Rightarrow \mathfrak{C}^{\delta+1}|_{\mathcal{L}_F} \sqsubseteq \mathfrak{T}_G^{\delta+2}|_{\mathcal{L}_F}$ *where* $G = \widetilde{inj}(m)\|_{\mathcal{L}_A}$

Proof. By transfinite induction on δ:

Case $\delta = 0$**:** It suffices to show that for some $\ell \in \mathcal{L}_F$, $\mathfrak{C}^1[\ell] = C(\bot_{\overline{R}})[\ell] = I_\top(\widetilde{inj}(\bot_{\overline{R}}))[\vec{\ell}] \sqsubseteq \mathfrak{T}_G^2[\ell] = I_\top(\mathfrak{T}_G^1)[\ell]$. This follows from the monotonicity of $I_\top$ if we show that $\widetilde{inj}(\bot_{\overline{R}}) \sqsubseteq \mathfrak{T}_G^1$. At points in $\mathcal{L}_F$ this is immediate, as $\widetilde{inj}(\bot_{\overline{R}})[\vec{\ell}' \in \mathcal{L}_F] = \bot_{\widetilde{S}} \sqsubseteq \mathfrak{T}_G^1[\vec{\ell}']$. At some point $\vec{\ell}' \in \mathcal{L}_A$, observe that $\widetilde{inj}(\bot_{\overline{R}})[\vec{\ell}'] = \widetilde{\tau}(\bot_{\widehat{S}})$, and by Lemma 20 we have that $\mathfrak{T}_G^1[\vec{\ell}'] = G[\vec{\ell}'] = \widetilde{inj}(m)\|_{\mathcal{L}_A}[\vec{\ell}'] = \widetilde{\tau}(m[\vec{\ell}'])$, whence the result holds from the monotonicity of $\widetilde{\tau}$.

Case $\delta+1$**:** It suffices to show that for some $\ell \in \mathcal{L}_F$ we have $\mathfrak{C}^{\delta+2}[\ell] = I_\top(\widetilde{inj}(\mathfrak{C}^{\delta+1}))[\ell] \sqsubseteq \mathfrak{T}_m^{\delta+3}[\ell] = I_\top(\mathfrak{T}_m^{\delta+2})[\ell]$, which holds if we can prove that $\widetilde{inj}(\mathfrak{C}^{\delta+1}) \sqsubseteq \mathfrak{T}_G^{\delta+2}$.

As the sequence $\mathfrak{C}$ is increasing, we have that $\mathfrak{C}^{\delta} \sqsubseteq \mathfrak{C}^{\delta+1}$, whence by transitivity we have that $\mathfrak{C}^{\delta}|_{\mathcal{L}_A} \sqsubseteq \mathfrak{m}|_{\mathcal{L}_A}$, thus by the inductive hypothesis, we have $\mathfrak{C}^{\delta+1}|_{\mathcal{L}_F} \sqsubseteq \mathfrak{T}_G^{\delta+2}|_{\mathcal{L}_F}$.

To establish the required inequality, we proceed by subcases on the partition of some arbitrary $\vec{\ell}'$:

Subcase $\vec{\ell}' \in \mathcal{L}_F$: $\widetilde{\mathsf{inj}}(\mathfrak{C}^{\delta+1})[\vec{\ell}'] = \mathfrak{C}^{\delta+1}[\vec{\ell}'] \sqsubseteq \mathfrak{T}_G^{\delta+2}|_{\mathcal{L}_F}[\vec{\ell}'] = \mathfrak{T}_G^{\delta+2}[\vec{\ell}']$, where the inequality holds from the application of the inductive hypothesis.

Subcase $\vec{\ell}' \in \mathcal{L}_A$: $\widetilde{\mathsf{inj}}(\mathfrak{C}^{\delta+1})[\vec{\ell}'] = \tilde{\tau}(\mathfrak{C}^{\delta+1}[\vec{\ell}']) \sqsubseteq \tilde{\tau}(\mathfrak{m}[\vec{\ell}']) = \widetilde{\mathsf{inj}}(\mathfrak{m})[\vec{\ell}'] = \mathsf{G}[\vec{\ell}'] = \mathfrak{T}_G^{\delta+2}[\vec{\ell}']$ where the final equality follows from Lemma 20, and the inequality follows from the assumption $\mathfrak{C}^{\delta+1}|_{\mathcal{L}_A} \sqsubseteq \mathfrak{m}|_{\mathcal{L}_A}$ and the monotonicity of $\tilde{\tau}$.

Case $\delta = \lambda$: If we show that $\widetilde{\mathsf{inj}}(\mathfrak{C}^{\lambda}) \sqsubseteq \mathfrak{T}_G^{\lambda}$, then we will have, for all $\vec{\ell} \in \mathcal{L}_F$:

$$\mathfrak{C}^{\lambda+1}[\vec{\ell}] = \mathsf{I}_{\top}(\widetilde{\mathsf{inj}}(\mathfrak{C}^{\lambda}))[\vec{\ell}] \sqsubseteq \mathsf{I}_{\top}(\mathfrak{T}_G^{\lambda})[\vec{\ell}] = \mathfrak{T}_G^{\lambda+1}[\vec{\ell}] \sqsubseteq \mathfrak{T}_G^{\lambda+2}[\vec{\ell}]$$

Expanding definitions, we must therefore show that

$$\widetilde{\mathsf{inj}}(\bigsqcup_{\beta<\lambda} \mathfrak{C}^{\beta}) \sqsubseteq \bigsqcup_{\beta<\lambda} \mathfrak{T}_G^{\beta} = \mathfrak{T}_G^{\lambda}$$

We show the inequality by cases:

Subcase $\vec{\ell} \in \mathcal{L}_A$:

$$\widetilde{\mathsf{inj}}(\bigsqcup_{\beta<\lambda} \mathfrak{C}^{\beta})[\vec{\ell}] = \tilde{\tau}(\bigsqcup_{\beta<\lambda} \mathfrak{C}^{\beta}[\vec{\ell}]) = \tilde{\tau}(\mathfrak{C}^{\lambda}[\vec{\ell}]) \sqsubseteq \tilde{\tau}(\mathfrak{m}[\vec{\ell}]) = \widetilde{\mathsf{inj}}(\mathfrak{m})\|_{\mathcal{L}_A}[\vec{\ell}] = \mathfrak{T}_G^{\lambda}[\vec{\ell}]$$

Where the final equality again follows from Lemma 20.

Subcase $\vec{\ell} \in \mathcal{L}_F$: It suffices to show that:

$$\widetilde{\mathsf{inj}}(\bigsqcup_{\beta<\lambda} \mathfrak{C}^{\beta})[\vec{\ell}] = \bigsqcup_{\beta<\lambda} \mathfrak{C}^{\beta}[\vec{\ell}] = \bot \sqcup \bigsqcup_{\beta<\lambda} \mathfrak{C}^{\beta+1}[\vec{\ell}] = \bigsqcup_{\beta<\lambda} \mathfrak{C}^{\beta+1}[\vec{\ell}] \sqsubseteq \mathfrak{T}_G^{\lambda}[\vec{\ell}]$$

that is, $\mathfrak{T}_G^{\lambda}[\vec{\ell}]$ is an upper bound for every $\mathfrak{C}^{\beta+1}[\vec{\ell}]$

By the definition of least upper bounds, we have that $\forall\alpha < \lambda.\mathfrak{C}^{\alpha}|_{\mathcal{L}_A} \sqsubseteq \mathfrak{C}^{\lambda}|_{\mathcal{L}_A} \sqsubseteq \mathfrak{m}|_{\mathcal{L}_A}$, whence by the transfinite hypothesis we have

$$\begin{aligned}\forall\alpha < \lambda.\mathfrak{C}^{\alpha+1}[\vec{\ell}] = \mathfrak{C}^{\alpha+1}|_{\mathcal{L}_F}[\vec{\ell}] \sqsubseteq \mathfrak{T}_G^{\alpha+2}|_{\mathcal{L}_F}[\vec{\ell}] = \mathfrak{T}_G^{\alpha+2}[\vec{\ell}]\\ \sqsubseteq \bigsqcup_{\beta<\lambda} \mathfrak{T}_G^{\beta}[\vec{\ell}] = \mathfrak{T}_G^{\lambda}[\vec{\ell}]\end{aligned}$$

which shows the upper bound as required.

□

Now, to prove $\mathfrak{C}^{\epsilon} \sqsubseteq \mathfrak{M}^{\epsilon}$, we prove the following:

Lemma 22. $\forall\delta \in \mu.\mathfrak{C}^{\delta} \sqsubseteq \mathfrak{M}^{\delta}$

Proof. By transfinite induction on δ.

Case $\delta = 0$**:** Trivial, $\mathfrak{C}^{0} = \bot = \mathfrak{M}^{0}$

Case $\delta + 1$**:** It suffices to show that for any $\vec{\ell} \in \mathcal{L}_A$, $\mathfrak{C}^{\delta+1}[\vec{\ell}] \sqsubseteq \mathfrak{M}^{\delta+1}[\vec{\ell}]$ and similarly for $\vec{\ell} \in \mathcal{L}_F$.

Subcase $\ell \in \mathcal{L}_A$**:** We must show that:

$$C(\mathfrak{C}^{\delta})[\vec{\ell}] = \widehat{F}(\widehat{inj}(\mathfrak{C}^{\delta}))[\ell] \sqsubseteq \bigsqcup_{i<\omega} U^{i}(\widehat{inj}(\mathfrak{M}^{\delta}))[\vec{\ell}]$$

which follows from the induction hypothesis, the monotonicity of $\widehat{inj}$, and Lemma 19.

Subcase $\ell \in \mathcal{L}_F$**:** We must show that $\mathfrak{C}^{\delta+1}[\ell] \sqsubseteq [\bigsqcup_{i<\omega} T(\widetilde{inj}(\mathfrak{M}^{\delta})\|_{\mathcal{L}_A})^{i}(\bot)][\ell] = \mathfrak{T}_{\mathfrak{G}}^{\omega}[\ell]$, where $\mathfrak{G} = \widetilde{inj}(\mathfrak{M}^{\delta})\|_{\mathcal{L}_A}$. From the induction hypothesis, we have that $\mathfrak{C}^{\delta} \sqsubseteq \mathfrak{M}^{\delta}$, from which it follows that $\mathfrak{C}^{\delta}|_{\mathcal{L}_A} \sqsubseteq \mathfrak{M}^{\delta}|_{\mathcal{L}_A}$. Thus, by Lemma 21 we conclude that:

$$\mathfrak{C}^{\delta+1}[\ell] \sqsubseteq \mathfrak{T}_{\mathfrak{G}}^{\delta+2}[\ell] \sqsubseteq \mathfrak{T}_{\mathfrak{G}}^{\omega}[\ell] \text{ (by Lemma 10)}$$

Case $\delta = \lambda$**:** Follows immediately from the definition of least-upper bound, transitivity, and the fact that, $\forall \beta < \lambda . \mathfrak{C}^\beta \sqsubseteq \mathfrak{M}^\beta$.

□

Theorem 13. $\mathfrak{C}^\epsilon \sqsubseteq \mathfrak{M}^\epsilon$

Proof. Immediate from Lemma 22. □

B.3.3 $\mathfrak{M}^\epsilon \sqsubseteq \mathfrak{C}^\epsilon$

We first need the following lemma:

Lemma 23. $\forall i \in \mathbb{N}, X : \overline{R}. X \sqsubseteq \mathfrak{C}^\epsilon \Rightarrow T(\widetilde{inj}(X)\|_{\mathcal{L}_A})^i(\bot) \sqsubseteq \widetilde{inj}(\mathfrak{C}^\epsilon)$

Proof. By induction on i.

Case $i = 0$**:** Trivial

Case $i + 1$**:** We assume $T(\widetilde{inj}(X)\|_{\mathcal{L}_A})^i(\bot) \sqsubseteq \widetilde{inj}(\mathfrak{C}^\epsilon)$ and show that $T(\widetilde{inj}(X)\|_{\mathcal{L}_A})^{i+1}(\bot) \sqsubseteq \widetilde{inj}(\mathfrak{C}^\epsilon)$. At $\vec{\ell} \in \mathcal{L}_A$, by Lemma 20 we have that $T(\widetilde{inj}(X)\|_{\mathcal{L}_A})^{i+1}(\bot)[\vec{\ell}] = \widetilde{inj}(X)\|_{\mathcal{L}_A}[\vec{\ell}] = \widetilde{\tau}(X[\vec{\ell}]) \sqsubseteq \widetilde{\tau}(\mathfrak{C}^\epsilon[\vec{\ell}]) = \widetilde{inj}(\mathfrak{C}^\epsilon)[\vec{\ell}]$. It remains to show that the inequality holds at some $\vec{\ell} \in \mathcal{L}_F$. By expanding definitions we have: $T(\widetilde{inj}(X)\|_{\mathcal{L}_A})^{i+1}(\bot)[\vec{\ell}] = I_\top(T(\widetilde{inj}(X)\|_{\mathcal{L}_A})^i(\bot))[\vec{\ell}] \sqsubseteq I_\top(\widetilde{inj}(\mathfrak{C}^\epsilon))[\vec{\ell}] = C(\mathfrak{C}^\epsilon)[\vec{\ell}] = \widetilde{inj}(C(\mathfrak{C}^\epsilon))[\vec{\ell}] = \widetilde{inj}(\mathfrak{C}^\epsilon)[\vec{\ell}]$, where the inequality holds from the monotonicity of $I_\top$ and the induction hypothesis, and the final equality comes from the definition of $\mathfrak{C}^\epsilon$ as a fixed point of C.

□

Lemma 24. $\forall i \in \mathbb{N}, X : \overline{R}. X \sqsubseteq \mathfrak{C}^\epsilon \Rightarrow U^i(\widehat{inj}(X)) \sqsubseteq \widehat{inj}(\mathfrak{C}^\epsilon)$

Proof. By induction on i.

Case $i = 0$**:** Trivial, by the monotonicity of $\widehat{inj}$.

Case $i+1$**:** By a similar argument to that made in Lemma 23, except on the monotonicity of $\widehat{F}$.

□

We next prove the following theorem:

Theorem 14. $\forall \delta \in \mu . \mathfrak{M}^{\delta} \sqsubseteq \mathfrak{C}^{\epsilon}$

Proof. By transfinite induction on δ.

Case $\delta = 0$**:** Trivial as $\mathfrak{M}^{0} = \bot$.

Case $\delta + 1$**:** It sufficient to show that $\forall \vec{\ell} . \mathfrak{M}^{\delta+1}[\vec{\ell}] \sqsubseteq \mathfrak{C}^{\epsilon}[\vec{\ell}]$. We proceed by the partition of $\vec{\ell}$:

Subcase $\vec{\ell} \in \mathcal{L}_A$**:** $\mathfrak{M}^{\delta+1}[\vec{\ell}] = [\bigsqcup_{i<\omega} U^{i}(\widehat{inj}(\mathfrak{M}^{\delta}))][\vec{\ell}] \sqsubseteq \widehat{inj}(\mathfrak{C}^{\epsilon})[\vec{\ell}] = \mathfrak{C}^{\epsilon}[\vec{\ell}]$, where the inequality holds from using the inductive hypothesis and Lemma 24 to show that $\widehat{inj}(\mathfrak{C}^{\epsilon})[\vec{\ell}]$ is an upper bound and thus greater than the least upper bound.

Subcase $\ell \in \mathcal{L}_F$**:** By a similar argument to the $\mathcal{L}_A$ case, but using Lemma 23.

Case $\delta = \lambda$**:** : By the transfinite hypothesis, $\forall \alpha < \lambda . \mathfrak{M}^{\alpha} \sqsubseteq \mathfrak{C}^{\epsilon}$, thus: $\bigsqcup_{\alpha<\lambda} \mathfrak{M}^{\alpha} \sqsubseteq \mathfrak{C}^{\epsilon}$.

□

Corollary 14.1. $\mathfrak{M}^{\epsilon} \sqsubseteq \mathfrak{C}^{\epsilon}$

Proof. Follows immediately from Theorem 14 and the definition of $\mathfrak{M}^{\epsilon}$ as the limit of the sequence $\mathfrak{M}^{\delta \in \mu}$. □

B.3.4 Proof of Theorem 10

Immediate from the definitions of *lfp* $M = \mathfrak{M}^{\epsilon}$ and *lfp* $C = \mathfrak{C}^{\epsilon}$, anti-symmetry, Theorem 13, and Corollary 14.1.

B.3.5 Fixpoints in the Abstract Interpreter

If $\widehat{F}$ is upper-ω-continuous, then the sequence $U^i(\widehat{inj}(\mathfrak{M}^\delta))$ converges to a fixpoint for all δ. To show this result, it is sufficient to show that the sequence $U^i(\widehat{inj}(\mathfrak{M}^\delta))$ is increasing for all δ. The result will then follow from a modified version of Kleene's fixpoint theorem. In the following, $\widehat{\mathfrak{M}^\delta}$ denotes $\widehat{inj}(\mathfrak{M}^\delta)$.

Lemma 25. $\forall\delta, k. U^k(\widehat{\mathfrak{M}^\delta}) \sqsubseteq \widehat{\mathfrak{M}^{\delta+1}}$

Proof. By cases. If $k = 0$, then the result is immediate, as $\mathfrak{M}^\delta$ is increasing, $\widehat{inj}$ is monotone.

For $k > 0$, we establish the result pointwise. At $\vec{\ell} \in \mathcal{L}_F$, $U^k(\widehat{\mathfrak{M}^\delta})[\vec{\ell}] = \widehat{\mathfrak{M}^\delta}[\vec{\ell}]$, whence the inequality holds by the reasoning above. At $\vec{\ell} \in \mathcal{L}_A$, we have:

$$U^k(\widehat{\mathfrak{M}^\delta})[\vec{\ell}] \sqsubseteq \bigsqcup_i U^i(\widehat{\mathfrak{M}^\delta})[\vec{\ell}] = \mathfrak{M}^{\delta+1}[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^{\delta+1})[\vec{\ell}]$$

□

Next define the following monotone function $PF : \overline{R} \to \overline{R}$:

$$PF = \lambda\langle\widetilde{m}, \widehat{m}\rangle.\langle\widetilde{m}, U(\widehat{inj}(\langle\widetilde{m}, \widehat{m}\rangle))|_{\mathcal{L}_A}\rangle$$

Lemma 26. $\forall\delta.\mathfrak{M}^\delta \sqsubseteq PF(\mathfrak{M}^\delta)$

Proof. By transfinite induction.

Case $\delta = 0$**:** Trivial.

Case $\delta + 1$**:** The inequality is immediate at all points in $\mathcal{L}_F$. We must therefore establish the inequality at some $\vec{\ell} \in \mathcal{L}_A$.

As $\mathfrak{M}^{\delta+1}[\vec{\ell}] = \bigsqcup_i U^i(\widehat{\mathfrak{M}^\delta})[\vec{\ell}]$, it suffices to show that for all i, $U^i(\widehat{\mathfrak{M}^\delta})[\vec{\ell}] \sqsubseteq U(\widehat{\mathfrak{M}^{\delta+1}})[\vec{\ell}] = PF(\mathfrak{M}^{\delta+1})[\vec{\ell}]$. We consider the form of i.

Subcase $i = 0$**:** we must show that $\widehat{\mathfrak{M}^{\delta}}[\vec{\ell}] = \mathfrak{M}^{\delta}[\vec{\ell}] \sqsubseteq U(\widehat{\mathfrak{M}^{\delta+1}})[\vec{\ell}]$. By the induction hypothesis, we have that $\mathfrak{M}^{\delta} \sqsubseteq \mathsf{PF}(\mathfrak{M}^{\delta})$, whence we have $\mathfrak{M}^{\delta}[\vec{\ell}] \sqsubseteq \mathsf{PF}(\mathfrak{M}^{\delta})[\vec{\ell}] = U(\widehat{\mathsf{inj}}(\mathfrak{M}^{\delta}))[\vec{\ell}] \sqsubseteq U(\widehat{\mathsf{inj}}(\mathfrak{M}^{\delta+1}))[\vec{\ell}]$, from the monotonicity of U and $\widehat{\mathsf{inj}}$ and the fact that $\mathfrak{M}^{\delta}$ is inreasing.

Subcase $i > 0$**:** Then $i = k + 1$ for some k, whence we must show $U^{i}(\widehat{\mathfrak{M}^{\delta}})[\vec{\ell}] = U(U^{k}(\widehat{\mathfrak{M}^{\delta}}))[\vec{\ell}] \sqsubseteq U(\widehat{\mathfrak{M}^{\delta+1}})[\vec{\ell}]$. By the monotonicity of U, it suffices to show that $U^{k}(\widehat{\mathfrak{M}^{\delta}}) \sqsubseteq \widehat{\mathfrak{M}^{\delta+1}}$, which holds from Lemma 25.

Case $\delta = \lambda$**:** By the induction hypothesis, $\mathfrak{M}^{\lambda}$ is defined as the least upper bound of post-fixed points of PF, whence $\mathfrak{M}^{\lambda}$ is also a post-fixed point of PF.

□

Theorem 15. $\forall \delta \in \mathsf{Ord}$, *the sequence* $U^{i}(\widehat{\mathfrak{M}^{\delta}})$ *is increasing.*

Proof. We will prove the sequence increasing by induction on i for some arbitrary δ.

Case $i = 0$**:** The inequality is immediate at $\vec{\ell} \in \mathcal{L}_F$. At $\vec{\ell} \in \mathcal{L}_A$, we have: $\widehat{\mathsf{inj}}(\mathfrak{M}^{\delta})[\vec{\ell}] = \mathfrak{M}^{\delta}[\vec{\ell}] \sqsubseteq \mathsf{PF}(\mathfrak{M}^{\delta})[\vec{\ell}] = U(\widehat{\mathsf{inj}}(\mathfrak{M}^{\delta}))[\vec{\ell}] = U^{1}(\widehat{\mathfrak{M}^{\delta}})[\vec{\ell}]$ by Lemma 26.

Case $i + 1$**:** Immediate from the monotonicity of U and the induction hypothesis.

□

B.4 *Formalisms, Soundness, and Termination of Widening Iteration*

We first define the instrumented semantic functions and extend the iteration functions defined in Appendix B.3:

$$\widehat{F}_{\triangledown}(X) = \begin{cases} X[\vec{\ell}]\,\widehat{\triangledown}\,\widehat{F}(X)[\vec{\ell}] & \vec{\ell} \in \mathcal{W}_A \\ \widehat{F}(X)[\vec{\ell}] & o.w. \end{cases} \qquad I^{\triangledown}_{\top}(X) = \begin{cases} X[\vec{\ell}]\,\widetilde{\triangledown}\,I_{\top}(X)[\vec{\ell}] & \vec{\ell} \in \mathcal{W}_F \\ I_{\top}(X)[\vec{\ell}] & o.w. \end{cases}$$

$$T_{\triangledown}(m)(X) = m\|_{\mathcal{L}_A} \sqcup I^{\triangledown}_{\top}(X)\|_{\mathcal{L}_F} \qquad U_{\triangledown}(X) = X\|_{\mathcal{L}_F} \sqcup \widehat{F}_{\triangledown}(X)\|_{\mathcal{L}_A}$$

$$M_{\triangledown}(X) = \Big\langle \bigsqcup_{i<\omega} U^{i}_{\triangledown}(\widehat{inj}(X))|_{\mathcal{L}_A},\ X|_{\mathcal{L}_F} \sqcup \bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(X)|_{\mathcal{L}_A})^{i}(\bot)|_{\mathcal{L}_F} \Big\rangle$$

We recall that $\widehat{\triangledown}$ is a widening operator on abstract states provided by the analysis, and $\widetilde{\triangledown}$ is the widening operator on mostly-concrete states, defined as

$$\langle m_f, m_a \rangle \widetilde{\triangledown} \langle m'_f, m'_a \rangle = \Big\langle \Big(\lambda x : X_f. \begin{cases} m_f & m'_f[x] \sqsubseteq m_f[x] \\ \top_F & o.w. \end{cases}\Big),\ \big(\lambda x : X_a. m_a[x] \triangledown_{\widehat{A}} m'_a[x]\big) \Big\rangle$$

where $\triangledown_{\widehat{A}}$ is a widening operator on values from $\widehat{A}$ provided by the analysis. We assume that $\bot \triangledown_{\widehat{A}} x = x$ and similarly that $\bot \widehat{\triangledown} x = x$. We use the following definition of widening operators. Both $\widehat{\triangledown}$ and $\triangledown_{\widehat{A}}$ must be upper bound operators and further for any increasing sequence x_i, the sequence $x^{\triangledown}_0 = x_0; x^{\triangledown}_{i+1} = x^{\triangledown}_i \triangledown x_{i+1}$ must converge in finite time (and similarly for $\triangledown_{\widehat{A}}$). We say an infinite sequence x_i converges in finite time if there is some index k such that $\forall j > k. x_j = x_k$.

These functions mirror the definitions in Appendix B.3, with two key differences. First, $\widehat{F}_{\triangledown}$ and $I^{\triangledown}_{\top}$ apply the widening operator at the widening points, which ensures termination (as we prove below). In addition, $M_{\triangledown}$ extends the mostly-concrete iteration term to join the results of mostly-concrete iteration with the input states. This joining ensures that the iteration is increasing, which is a key condition of our termination proofs.

B.4.1 Soundness

We first prove the soundness of the widening version of subfixpoint iteration. Define a sequence $\mathfrak{M}_{\triangledown}^{\delta\in\mu}$ as:

$$\mathfrak{M}_{\triangledown}^{0} = \bot \qquad \mathfrak{M}_{\triangledown}^{\delta+1} = M_{\triangledown}(\mathfrak{M}_{\triangledown}^{\delta}) \qquad \mathfrak{M}_{\triangledown}^{\lambda} = \bigsqcup_{\beta<\lambda} \mathfrak{M}_{\triangledown}^{\beta}$$

We have shown that this sequence over-approximates the sequence $\mathfrak{M}^{\delta}$ as follows:

Theorem 16. $\forall\delta.\mathfrak{M}^{\delta} \sqsubseteq \mathfrak{M}_{\triangledown}^{\delta}$

Lemma 27. $\forall i \in \mathbb{N}, m, m' : \widetilde{R}.m \sqsubseteq m' \Rightarrow T(m)^{i}(\bot) \sqsubseteq T_{\triangledown}(m')^{i}(\bot)$

Proof. By induction on i.

Case $i = 0$**:** Trivial

Case $i+1$**:** For the values at $\vec{\ell} \in \mathcal{L}_A$, the inequality is immediate by the assumption on m and m'. It remains to show that the inequality holds at some arbitrary $\vec{\ell} \in \mathcal{L}_F$. For $\vec{\ell} \in \mathcal{W}_F$, we have by the induction hypothesis $T(m)^{i}(\bot) \sqsubseteq T_{\triangledown}(m')^{i}(\bot)$ whence from the monotonicity of $I_{\top}$ we therefore have $I_{\top}(T(m)^{i}(\bot)) \sqsubseteq I_{\top}(T_{\triangledown}(m')^{i}(\bot))$, and thus: $T(m)^{i+1}(\bot)[\vec{\ell}] = I_{\top}(T(m)^{i}(\bot))[\vec{\ell}] \sqsubseteq I_{\top}(T_{\triangledown}(m')^{i}(\bot))[\vec{\ell}] \sqsubseteq T_{\triangledown}(m')^{i}(\bot)[\vec{\ell}]\widetilde{\triangledown}I_{\top}(T_{\triangledown}(m')^{i}(\bot))[\vec{\ell}] = T_{\triangledown}(m')^{i+1}(\bot)$, where the inequality holds as $\widetilde{\triangledown}$ is an upper bound operator. For $\vec{\ell} \notin \mathcal{W}_F$, then the result holds from the induction hypothesis and the monotonicity of $I_{\top}$.

□

Lemma 28. $\forall i \in \mathbb{N}, m, m' : \widehat{R}.m \sqsubseteq m' \Rightarrow U^{i}(m) \sqsubseteq U_{\triangledown}^{i}(m')$

Proof. By symmetric reasoning to that in Lemma 27, except using the monotonicity of $\widehat{F}$ and that $\widehat{\triangledown}$ is an upper bound operator. □

Proof of Theorem 16. By transfinite induction. In the inductive step, the inductive hypothesis, monotonicity of $\widehat{\mathrm{inj}}$ and $\widetilde{\mathrm{inj}}$, and Lemmas 27 and 28 give that the subfixpoint terms of $\mathfrak{M}^{\delta+1}$ are smaller than those in $\mathfrak{M}_{\triangledown}^{\delta+1}$ giving the desired result. The limit case follows from the definition of least upper bounds. □

As a direct corollary of this theorem, we have: $\mathit{lfp}\, C = \mathit{lfp}\, M = \mathfrak{M}^{\epsilon} \sqsubseteq \mathfrak{M}_{\triangledown}^{\epsilon}$. If we show that if the sequence $\mathfrak{M}_{\triangledown}^{\delta}$ converges in a finite number of k steps to a fixed-point of $M_{\triangledown}$, we will then have: $\mathit{lfp}\, C = \mathit{lfp}\, M = \mathfrak{M}^{\epsilon} \sqsubseteq \mathfrak{M}_{\triangledown}^{\epsilon} = \mathfrak{M}_{\triangledown}^{k}$, i.e., $\alpha_F(\mathit{lfp}\, F) \sqsubseteq \mathfrak{M}_{\triangledown}^{k}$.

B.4.2 Termination

We first show that $\forall i \in \mathbb{N}$, the sequence $\mathfrak{M}_{\triangledown}^{i}$ is increasing.

Lemma 29. $\forall j. \mathfrak{M}_{\triangledown}^{j} \sqsubseteq \mathfrak{M}_{\triangledown}^{j+1}$

Proof. Consider an arbitrary j. We establish the inequality pointwise:

Case $\vec{\ell} \in \mathcal{L}_F$: $\mathfrak{M}_{\triangledown}^{j}[\vec{\ell}] = \mathfrak{M}_{\triangledown}^{j}|_{\mathcal{L}_F}[\vec{\ell}] \sqsubseteq \mathfrak{M}_{\triangledown}^{j}|_{\mathcal{L}_F}[\vec{\ell}] \sqcup \bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{\mathrm{inj}}(\mathfrak{M}_{\triangledown}^{j})\|_{\mathcal{L}_A})^{i}(\bot)|_{\mathcal{L}_F}[\vec{\ell}] = \mathfrak{M}_{\triangledown}^{j+1}[\vec{\ell}]$

Case $\vec{\ell} \in \mathcal{L}_A$: $\mathfrak{M}_{\triangledown}^{j}[\vec{\ell}] = \widehat{\mathrm{inj}}(\mathfrak{M}_{\triangledown}^{j})[\vec{\ell}] = U_{\triangledown}^{0}(\widehat{\mathrm{inj}}(\mathfrak{M}_{\triangledown}^{j}))[\vec{\ell}] \sqsubseteq \bigsqcup_{i<\omega} U_{\triangledown}^{i}(\widehat{\mathrm{inj}}(\mathfrak{M}_{\triangledown}^{j}))[\vec{\ell}] = \mathfrak{M}_{\triangledown}^{j+1}[\vec{\ell}]$

□

We next show that every subfixpoint iteration in the abstract interpreter terminates. Define the following family of sequences indexed by n:

$$\mathfrak{P}_{(n,j)} = U_{\triangledown}^{j}(\widehat{\mathrm{inj}}(\mathfrak{M}_{\triangledown}^{n}))$$

We need the following lemma.

Lemma 30.

1. $\forall n. U_{\triangledown}^{n}(\widehat{\mathrm{inj}}(\mathfrak{M}_{\triangledown}^{n})) \sqsubseteq \widehat{\mathrm{inj}}(\mathfrak{M}_{\triangledown}^{n+1})$

2. *The sequence* $U_{\triangledown}^{j}(\widehat{\mathrm{inj}}(\mathfrak{M}_{\triangledown}^{0} = \bot_{\overline{R}}))$ *is increasing.*

3. *For all* n *and* $\vec{\ell} \in \mathcal{L}_A$*, if all sequences* $\mathfrak{P}_{(m<n,j)}$ *are increasing and converge in a finite number of steps, then* $\widehat{inj}(\mathfrak{M}^n_\triangledown)[\vec{\ell}] = \bot_{\widehat{S}}$ *or* $\exists k < n, j > 0.\widehat{inj}(\mathfrak{M}^n_\triangledown)[\vec{\ell}] = U^j_\triangledown(\widehat{inj}(\mathfrak{M}^k_\triangledown))[\vec{\ell}]$.

4. *For all* n*, if all sequences* $\mathfrak{P}_{(m\leqslant n,j)}$ *are increasing and converge in a finite number of steps, then the sequence* $U^j_\triangledown(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))$ *is increasing.*

Proof.

1. At $\vec{\ell} \in \mathcal{L}_F$, we have by a simple inductive argument that $U^n_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^n_\triangledown)[\vec{\ell}] \sqsubseteq \widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}]$ from Lemma 29. At $\vec{\ell} \in \mathcal{L}_A$, we have: $U^n_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}] \sqsubseteq \bigsqcup_{i<\omega} U^i_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}] = \mathfrak{M}^{n+1}_\triangledown[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}]$.

2. By straightforward induction on j, using the fact that $\widehat{F}$ is monotone and that $\widehat{\triangledown}$ is an upper bound operator. The base case holds because at $\vec{\ell} \in \mathcal{L}_A$, $\widehat{inj}(\bot_{\overline{R}})[\vec{\ell}] = \bot_{\widehat{S}}$ and for $\vec{\ell} \in \mathcal{L}_F$ $\forall i.U^i_\triangledown(\widehat{inj}(\bot_{\overline{R}}))[\vec{\ell}] = \widehat{\tau}(\bot_{\widetilde{S}})$.

3. By induction on n.

   Case $n = 0$**:** Then $\widehat{inj}(\mathfrak{M}^0_\triangledown)[\vec{\ell}] = \bot_{\overline{R}}[\vec{\ell}] = \bot_{\widehat{S}}$, trivially giving the desired result.

   Case $n + 1$**:** Then $\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}] = \bigsqcup_{i<\omega} U^i_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}]$. By hypothesis, the sequence $U^i_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))$ is increasing and converges in a finite number of steps, whence $\exists p. \bigsqcup_{i<\omega} U^i_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}] = U^p_\triangledown(\widehat{inj}(\mathfrak{M}^n_\triangledown))[\vec{\ell}]$. If $p = 0$, then the inductive hypothesis gives us the desired result (it is immediate that if the sequences $\mathfrak{P}_{(m<n+1,j)}$ are terminating increasing sequences, then the sequences $\mathfrak{P}_{(m<n,j)}$ must be as well). Otherwise, we take $k = n, j = p$ completing the proof.

4. By induction on j.

   Case $j = 0$**:** We must show that $\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown) \sqsubseteq U_\triangledown(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))$. By cases:

   Subcase $\vec{\ell} \in \mathcal{W}_A$**:** $U_\triangledown(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))[\vec{\ell}] = \widehat{F}_\triangledown(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^{n+1}_\triangledown)[\vec{\ell}]\widehat{\triangledown}\widehat{F}(\widehat{inj}(\mathfrak{M}^{n+1}_\triangledown))[\vec{\ell}]$, and we have the required inequality from $\widehat{\triangledown}$ being an upper bound operator.

Subcase $\vec{\ell} \in \mathcal{L}_A \wedge \vec{\ell} \notin \mathcal{W}_A$**:** We must show that

$$\widehat{inj}(\mathfrak{M}_{\triangledown}^{n+1})[\vec{\ell}] \sqsubseteq U_{\triangledown}(\widehat{inj}(\mathfrak{M}_{\triangledown}^{n+1}))[\vec{\ell}] = \widehat{F}(\widehat{inj}(\mathfrak{M}_{\triangledown}^{n+1}))[\vec{\ell}]$$

By hypothesis, the sequences $\mathfrak{P}_{(m \leqslant n,j)}$ are increasing and converge in finite number of steps. As $m \leqslant n$ implies $m < n+1$, we can satisfy the hypothesis of part (3) for $\widehat{inj}(\mathfrak{M}_{\triangledown}^{n+1})$. Suppose then that $\widehat{inj}(\mathfrak{M}_{\triangledown}^{n+1})[\vec{\ell}] = \bot$. Then we trivially have $\widehat{inj}(\mathfrak{M}_{\triangledown}^{n+1})[\vec{\ell}] = \bot \sqsubseteq \widehat{F}(\widehat{inj}(\mathfrak{M}_{\triangledown}^{n+1}))[\vec{\ell}]$. Next, suppose that there exists some $k < n+1, j > 0$ such that $\widehat{inj}(\mathfrak{M}_{\triangledown}^{n+1})[\vec{\ell}] = U_{\triangledown}^{j}(\widehat{inj}(\mathfrak{M}_{\triangledown}^{k}))[\vec{\ell}]$. Expanding the definition of $U_{\triangledown}$, we must then show that $U_{\triangledown}^{j}(\widehat{inj}(\mathfrak{M}_{\triangledown}^{k}))[\vec{\ell}] = \widehat{F}(U_{\triangledown}^{j-1}(\widehat{inj}(\mathfrak{M}_{\triangledown}^{k})))[\vec{\ell}] \sqsubseteq \widehat{F}(\widehat{inj}(\mathfrak{M}_{\triangledown}^{n}))[\vec{\ell}]$. By part (1) and Lemma 29, we have that $U_{\triangledown}^{j-1}(\widehat{inj}(\mathfrak{M}_{\triangledown}^{k})) \sqsubseteq \widehat{inj}(\mathfrak{M}_{\triangledown}^{k+1}) \sqsubseteq \widehat{inj}(\mathfrak{M}_{\triangledown}^{n+1})$, giving the desired result via monotonicity of $\widehat{F}$.

Subcase $\vec{\ell} \in \mathcal{L}_F$**:** Trivial.

Case $j+1$**:** Immediate at $\vec{\ell} \in \mathcal{W}_A$ from the definition of $\widehat{\triangledown}$ as an upper bounds operator, trivial at $\vec{\ell} \in \mathcal{L}_F$, and from the induction hypothesis and the monotonicity of $\widehat{F}$ for $\vec{\ell} \in \mathcal{L}_A \wedge \vec{\ell} \notin \mathcal{W}_A$.

□

We are now ready to prove that, for any n, $\mathfrak{P}_{(n,j)}$ converges in finite steps, i.e., the abstract iteration terminates.

Theorem 17. $\forall n.\mathfrak{P}_{(n,j)}$ *is an increasing sequence that converges in a finite number of steps.*

Proof. By strong induction on n.

Case $n = 0$**:** Lemma 30 part (2) gives us that the sequence is increasing. To show that the sequence converges in a finite number of steps, it suffices to show that all labels in the widening set converge in a finite number of steps, whence the overall termination will follow from the definiton of widening sets. Consider an arbitrary

$\vec{\ell} \in \mathcal{W}_A$, and the values computed at each element of the sequence $\mathfrak{P}_{(0,j)}$.

$$\begin{aligned}\mathfrak{P}_{(0,0)}[\vec{\ell}] &= \widehat{inj}(\mathfrak{M}^0_\triangledown)[\vec{\ell}] = \widehat{inj}(\bot_{\overline{R}})[\vec{\ell}] = \bot_{\widehat{S}} \\ \mathfrak{P}_{(0,i+1)}[\vec{\ell}] &= U_\triangledown(U^i_\triangledown(\widehat{inj}(\bot_{\overline{R}})))[\vec{\ell}] = \widehat{F}_\triangledown(U^i_\triangledown(\widehat{inj}(\bot_{\overline{R}})))[\vec{\ell}] \\ &= U^i_\triangledown(\widehat{inj}(\bot_{\overline{R}}))[\vec{\ell}]\widehat{\triangledown}\widehat{F}(U^i_\triangledown(\widehat{inj}(\bot_{\overline{R}})))[\vec{\ell}] \\ &= \mathfrak{P}_{(0,i)}[\vec{\ell}]\widehat{\triangledown}\widehat{F}(U^i_\triangledown(\widehat{inj}(\bot_{\overline{R}})))[\vec{\ell}]\end{aligned}$$

By the definition of widening operators, this sequence will converge in a finite number of steps (that the sequence $\bot_{\widehat{S}}, \widehat{F}(\widehat{inj}(\bot_{\overline{R}})), \ldots, \widehat{F}(U^n_\triangledown(\widehat{inj}(\bot_{\overline{R}})))$ is increasing follows from Lemma 30 part (2) and the monotonicity of $\widehat{F}$). Thus, all widening points will converge in a finite number of steps, implying that $\mathfrak{P}_{(0,j)}$ converges.

Case $n+1$**:** We assume that all sequences $\mathfrak{P}_{(m<n+1,j)}$ are increasing and converge in a finite number of steps. From Lemma 30 part (4), $\mathfrak{P}_{(n+1,j)}$ is increasing. To show that $\mathfrak{P}_{(n+1,j)}$ converges in a finite number of steps, we will again show that any arbitrary widening point $\vec{\ell}$ converges in a finite number of steps. Define a sequence finitization operator $\lfloor \mathfrak{S} \rfloor = \langle s_0, \ldots, s_{k-1} \rangle$ where $s_i = \mathfrak{S}_{i+1}$ and k is defined as the smallest natural number such that:

$$k \geqslant 1 \qquad\qquad \forall m \geqslant k. \mathfrak{S}_m = \mathfrak{S}_k$$

In other words, the finitization operator extracts a sequence of at least length one, starting from the *second* element of the sequence, up to the first element after which the sequence repeats. This operation is undefined on sequences that do not converge in finite steps.

Define now the following sequence:

$$\mathfrak{L} = \bowtie_{m<n+1} \lfloor \mathfrak{P}_{(m,j)}[\vec{\ell}] \rfloor \bowtie \mathfrak{P}_{(n+1,j\geqslant 1)}[\vec{\ell}]$$

where $\bowtie$ denotes sequence concatenation. The $\lfloor \mathfrak{P}_{(m,j)}[\vec{\ell}] \rfloor$ operation is well defined from the induction hypothesis that all such sequences converge in finite steps. As a result, $\mathfrak{L}$ has a finite prefix of values before reaching elements drawn from $\mathfrak{P}_{(n+1,j)}[\vec{\ell}]$. If we show that the sequence $\mathfrak{L}$ converges in a finite number of steps,

this will in turn imply that $\mathfrak{P}_{(n+1,j)}[\vec{\ell}]$ also converges in a finite number of steps. We now show that $\mathfrak{L}_{i>0} = \mathfrak{L}_{i-1}\widehat{\triangledown}\mathfrak{W}_i$ and $\mathfrak{L}_0 = \mathfrak{W}_0$, where $\mathfrak{W}_i$ is an increasing sequence. From the definition of widening operators, this sequence will converge and thus, as argued above, will prove convergence of the sequence $\mathfrak{P}_{(n+1,j)}[\vec{\ell}]$.

From the definition of $\mathfrak{L}$ it is immediate that:

$$\forall i.\mathfrak{L}_i \equiv \mathfrak{P}_{(m,k)}[\vec{\ell}] = U^k_\triangledown(\widehat{inj}(\mathfrak{M}^m_\triangledown))[\vec{\ell}] \text{ (for some } m \leqslant n+1 \text{ and } k \geqslant 1) \tag{B.3}$$

$$\forall i, k > 0, m \leqslant n+1.\mathfrak{L}_{i+1} \equiv U^{k+1}_\triangledown(\widehat{inj}(\mathfrak{M}^m_\triangledown))[\vec{\ell}] \Rightarrow \mathfrak{L}_i \equiv U^k_\triangledown(\widehat{inj}(\mathfrak{M}^m_\triangledown))[\vec{\ell}] \tag{B.4}$$

We will now show that for all $i > 0$, $\mathfrak{L}_{i>0}$ is defined as $\mathfrak{P}_{(m,k)}[\vec{\ell}]$ for some $m \leqslant n+1$ and $k \geqslant 1$, and further that $\mathfrak{L}_i = \mathfrak{L}_{i-1}\widehat{\triangledown}\widehat{F}(\mathfrak{P}_{(m,k-1)})[\vec{\ell}]$.

Consider an arbitrary $i > 0$. Then $\mathfrak{L}_i = U^k_\triangledown(\widehat{inj}(\mathfrak{M}^m_\triangledown))[\vec{\ell}], m \leqslant n+1, k > 0$ by Equation (B.3). Suppose $k \neq 1$. Then $k = j+1$, and

$$U^{j+1}_\triangledown(\widehat{inj}(\mathfrak{M}^m_\triangledown))[\vec{\ell}] = U^j_\triangledown(\widehat{inj}(\mathfrak{M}^m_\triangledown))[\vec{\ell}]\triangledown\widehat{F}(U^j_\triangledown(\widehat{inj}(\mathfrak{M}^m_\triangledown))[\vec{\ell}] = \mathfrak{L}_{i-1}\widehat{\triangledown}\widehat{F}(\mathfrak{P}_{(m,k-1)})[\vec{\ell}]$$

where the final equality holds from the definition of $\mathfrak{P}$ and (B.4) above.

Next, suppose that $k = 1$. Then

$$U_\triangledown(\widehat{inj}(\mathfrak{M}^m_\triangledown))[\vec{\ell}] = \widehat{inj}(\mathfrak{M}^m_\triangledown)[\vec{\ell}]\widehat{\triangledown}\widehat{F}(\widehat{inj}(\mathfrak{M}^m_\triangledown))[\vec{\ell}] = \mathfrak{M}^m_\triangledown[\vec{\ell}]\widehat{\triangledown}\widehat{F}(\widehat{inj}(\mathfrak{M}^m_\triangledown))[\vec{\ell}] = \mathfrak{M}^m_\triangledown[\vec{\ell}]\widehat{\triangledown}\widehat{F}(\mathfrak{P}_{(m,k-1)})[\vec{\ell}]$$

We must show that $\mathfrak{M}^m_\triangledown[\vec{\ell}] = \mathfrak{L}_{i-1}$. First, observe that m cannot be 0 by our assumption that $i > 0$.

As $i > 0$, there must be some preceding element $\mathfrak{L}_{i-1}$ and by definition this element is the final element of $\lfloor\mathfrak{P}_{(m-1,j)}\rfloor$. (This follows from the fact that $\mathfrak{P}_{(m,1)}[\vec{\ell}]$ must be the first element of one of the sequences that are concatenated together to form $\mathfrak{L}$.)

From the definition of $\lfloor\mathfrak{S}\rfloor$, this final element is $\mathfrak{P}_{(m-1,p)}[\vec{\ell}]$, where p is the first index greater than or equal to 1 after which the sequence $\mathfrak{P}_{(m-1,j)}[\vec{\ell}]$ repeats infinitely. Further, as the sequence $\mathfrak{P}_{(m-1,j)}$ is increasing by the induction hypothesis, we have:

$$\mathfrak{M}^m_\triangledown[\vec{\ell}] = \bigsqcup_{i<\omega}\mathfrak{P}_{(m-1,i)}[\vec{\ell}] = \mathfrak{P}_{(m-1,p)}[\vec{\ell}] = \mathfrak{L}_{i-1}$$

Finally, by expanding definitions we have:

$$\mathfrak{L}_0 = \mathfrak{P}_{(0,1)}[\vec{\ell}] = \widehat{F}_{\triangledown}(\widehat{inj}(\mathfrak{M}^0_{\triangledown}))[\vec{\ell}] = \widehat{inj}(\bot_{\overline{R}})[\vec{\ell}]\widehat{\triangledown}\widehat{F}(\widehat{inj}(\mathfrak{M}^0_{\triangledown}))[\vec{\ell}]$$
$$= \widehat{F}(U^0_{\triangledown}(\widehat{inj}(\mathfrak{M}^0_{\triangledown})))[\vec{\ell}] = \widehat{F}(\mathfrak{P}_{(0,0)})[\vec{\ell}]$$

We therefore define the sequence $\mathfrak{W}_i$ as:

$$\mathfrak{W}_0 = \mathfrak{L}_0 = \widehat{F}(\mathfrak{P}_{(0,0)})[\vec{\ell}]$$
$$\mathfrak{W}_{i+1} = \widehat{F}(\mathfrak{P}_{(m,k-1)})[\vec{\ell}] \text{ where } \mathfrak{L}_{i+1} \equiv \mathfrak{P}_{(m,k)}$$

It remains to show that this is an increasing sequence. From the definition of $\mathfrak{L}$, if $\mathfrak{W}_i \equiv \widehat{F}(\mathfrak{P}_{(m,k-1)})[\vec{\ell}]$ for some $m \leqslant n+1$ and k, then $\mathfrak{W}_{i+1} = \widehat{F}(\mathfrak{P}_{(m,k)})[\vec{\ell}]$ or $\mathfrak{W}_{i+1} = \widehat{F}(\mathfrak{P}_{(m+1,1)})[\vec{\ell}]$. By the monotonicity of $\widehat{F}$, it suffices to show that $\mathfrak{P}_{(m,k-1)} \sqsubseteq \mathfrak{P}_{(m,k)}$ and that $\mathfrak{P}_{(m,k-1)} \sqsubseteq \mathfrak{P}_{(m+1,1)}$. For the former, if $m < n+1$ the result holds from the induction hypothesis, otherwise if $m = n+1$ the result holds from Lemma 30 part (4). The latter is only possible if $m < n+1$. By Lemma 30 part (1), $\mathfrak{P}_{(m,k-1)} \sqsubseteq \mathfrak{P}_{(m+1,0)}$ for any m. It remains to show that $\mathfrak{P}_{(m+1,0)} \sqsubseteq \mathfrak{P}_{(m+1,1)}$. This holds by either the induction hypothesis ($m < n$) or Lemma 30 part (4) ($m = n$).

□

We next show that all iterations in the mostly-concrete interpreter converge in finite time.

Lemma 31.

1. $\forall m : \widetilde{R}.T_{\triangledown}(m)^i(\bot)$ *is increasing.*

2. $\forall m : \widetilde{R}.T_{\triangledown}(m)^i(\bot)$ *converges.*

Proof.

1. By induction on i. The in the inductive step, for flow edges in $\mathcal{W}_F$ the inequality is due to the $\widetilde{\triangledown}$ being an upper bound operator, whereas for flow edges not in $\mathcal{W}_F$ the result holds from the monotonicity of $I_{\top}$ and the IH.

2. As in Theorem 17, we consider an arbitrary widening point $\vec{\ell}$. Then the values computed at each step during subfixpoint iteration, $T_{\triangledown}(m)^i(\bot_{\widetilde{R}})[\vec{\ell}]$ form the following sequence:

$$T_{\triangledown}(m)^0(\bot_{\widetilde{R}})[\vec{\ell}] = \bot_{\widetilde{S}}$$
$$T_{\triangledown}(m)^{i+1}(\bot_{\widetilde{R}})[\vec{\ell}] = T_{\triangledown}(m)^i(\bot_{\widetilde{R}})[\vec{\ell}]\widetilde{\triangledown}I_{\top}(T_{\triangledown}(m)^i(\bot_{\widetilde{R}}))[\vec{\ell}]$$

The sequence $\bot_{\widetilde{S}}, I_{\top}(\bot_{\widetilde{R}})[\vec{\ell}], \ldots, I_{\top}(T_{\triangledown}(m)^i(\bot_{\widetilde{R}}))[\vec{\ell}]$ is increasing from the monotonicity of $I_{\top}$ and that $T_{\triangledown}(m)^i(\bot_{\widetilde{R}})$ is increasing proved in part (1) above. Thus the above sequence converges after a finite number of steps, giving us the termination result.

□

Lemma 32. *The sequence $\mathfrak{M}^i_{\triangledown}|_{\mathcal{L}_A}$ converges in a finite number of steps*

Proof. By Theorem 17, as each subfixpoint iteration is finite, we can construct a sequence similar to $\mathfrak{L}$ in the proof Theorem 17. Using similar reasoning, we show that this sequence stabilizes in a finite number of steps. This implies that there is some index k, such that when computing $\mathfrak{M}^m_{\triangledown}|_{\mathcal{L}_A}, m \geqslant k$, the values at all widening points in $\mathcal{W}_A$ will converge to the same values. This implies that after a k steps, the sequence $\mathfrak{M}^k_{\triangledown}|_{\mathcal{L}_A}$ stabilizes. □

Theorem 18. *The sequence $\mathfrak{M}^i_{\triangledown}$ converges in a finite number of steps.*

Proof. We first note that, as $I^{\triangledown}_{\top}$ is deterministic, if $m|_{\mathcal{L}_A} = m'|_{\mathcal{L}_A}$ then $\bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(m)\|_{\mathcal{L}_A})^i(\bot) = \bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(m')\|_{\mathcal{L}_A})^i(\bot)$ By Lemma 32, there is some k such that $\mathfrak{M}^k_{\triangledown}|_{\mathcal{L}_A} = \mathfrak{M}^{k+1}_{\triangledown}|_{\mathcal{L}_A} = \mathfrak{M}^{k+2}_{\triangledown}|_{\mathcal{L}_A}$. As $\mathfrak{M}^k_{\triangledown}|_{\mathcal{L}_A} = \mathfrak{M}^{k+1}_{\triangledown}|_{\mathcal{L}_A}$ we have that

$$\bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(\mathfrak{M}^k_{\triangledown})\|_{\mathcal{L}_A})^i(\bot) = \bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(\mathfrak{M}^{k+1}_{\triangledown})\|_{\mathcal{L}_A})^i(\bot)$$

Consider next some arbitrary $\vec{\ell} \in \mathcal{L}_F$. Let $\mathcal{U}_{k+1} = \bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(\mathfrak{M}^k_{\triangledown})\|_{\mathcal{L}_A})^i(\bot)[\vec{\ell}]$ and $\mathcal{U}_{k+2} = \bigsqcup_{i<\omega} T_{\triangledown}(\widetilde{inj}(\mathfrak{M}^{k+1}_{\triangledown})\|_{\mathcal{L}_A})^i(\bot)[\vec{\ell}]$. By the above equality, we have that $\mathcal{U}_{k+1} = \mathcal{U}_{k+2}$,

and thus:

$$\mathfrak{M}_{\triangledown}^{k+1}[\vec{\ell}] = \mathfrak{M}_{\triangledown}^{k}[\vec{\ell}] \sqcup \mathcal{U}_{k+1} = \mathfrak{M}_{\triangledown}^{k}[\vec{\ell}] \sqcup \mathcal{U}_{k+1} \sqcup \mathcal{U}_{k+1} =$$

$$(\mathfrak{M}_{\triangledown}^{k}[\vec{\ell}] \sqcup \mathcal{U}_{k+1}) \sqcup \mathcal{U}_{k+2} = \mathfrak{M}_{\triangledown}^{k+1}[\vec{\ell}] \sqcup \mathcal{U}_{k+2} = \mathfrak{M}_{\triangledown}^{k+2}[\vec{\ell}]$$

We therefore have $\mathfrak{M}_{\triangledown}^{k+1}|_{\mathcal{L}_A} = \mathfrak{M}_{\triangledown}^{k+2}|_{\mathcal{L}_A}$ and $\mathfrak{M}_{\triangledown}^{k+1}|_{\mathcal{L}_F} = \mathfrak{M}_{\triangledown}^{k+2}|_{\mathcal{L}_F}$, whence $\mathfrak{M}_{\triangledown}^{k+1} = \mathfrak{M}_{\triangledown}^{k+2} = M_{\triangledown}(\mathfrak{M}_{\triangledown}^{k+1})$. Thus, in $k+1$ steps, the sequence reaches a fixpoint of $M_{\triangledown}$, and thus the entire sequence stabilizes in finite time. $\square$

Appendix C
PROOFS FOR CHAPTER 7

This chapter contains initial proofs of correctness for the scheduling algorithm described in Chapter 7. In particular, it proves that loop inconsistent executions are impossible with our scheduling approach. It also contains an initial optimality argument. In many places we rely on natural language theorem statements and arguments: fully formalizing these theorems and proofs is left for future work.

General Notation We will denote by $a \xrightarrow{\pi} b$ a path π from a to b. For some path π, denote by $\pi(i)$ the ith element of the sequence, and by $|\pi|$ the length of the sequence π.

C.1 Loop Consistency

Associate with thread t the loop counter vector $\mathfrak{I}_t$ of length k, where k is the number of unique loop headers in a method. We will denote by $\mathfrak{I}^h$ the counter associated with the loop with header h in vector $\mathfrak{I}$ (we leave the mapping from headers to numerical positions unspecified). When a thread t steps from point a to point b, the new value value of $\mathfrak{I}_t^h$ becomes:

$$\iota \text{ if } a \in \mathcal{L}(h) \land b \notin \mathcal{L}(h) \land b \neq h \tag{C.1}$$

$$0 \text{ if } b = h \land \mathfrak{I}_t^h = \iota \tag{C.2}$$

$$\mathfrak{I}_t^h + 1 \text{ if } a = h \land b \in \mathcal{L}(h) \tag{C.3}$$

$$\mathfrak{I}_t^h \text{ otherwise} \tag{C.4}$$

Denote by $[\![a \to b]\!]$ the pointwise lifting of the above transformation to vectors, and $[\![\pi]\!]$ the composition of the pairwise edge transformers. We say a loop counter is *live*

(indicating a thread with that counter is actively executing the loop) if the counter is neither 0 nor ι.

An execution state is loop consistent if, for all distinct pairs of p and q, $\mathfrak{I}_p \sim \mathfrak{I}_q$ where $\mathfrak{I}_1 \sim \mathfrak{I}_2$ is defined as:

$$\forall h \in \mathrm{dom}(\mathfrak{I}_1).\mathfrak{I}_1^h = \mathfrak{I}_2^h \vee \mathfrak{I}_1 = \iota \vee \mathfrak{I}_2^h = \iota$$

C.2 Proofs

As in Chapter 7, we will abuse notation and use the name of a thread of execution to also denote the program location where it is paused.

We first review some helpful properties of loops and paths.

Lemma 33. *If there exists a cycle free path π from a to b that involves a backjump to h, then a is contained with a loop with header h.*

Proof. Let x be the predecessor to the backjump to h. Then x must be reachable from a. As the path is loop free, the path from a to x must not contain h. There must therefore be a path from a to h following the predecessors of a. If not, we could traverse the predecessor relation from a to method entry. Let this (hypothetical) path without h from the entry to a be π'. If this path existed, we could extend π' with the h-free path from a to x, contradicting our (implicit) assumption that x is dominated by h. □

Lemma 34. *If every loop free path π from a loop header h to x traverses a backjump, then $x \notin \mathcal{L}(h)$.*

Proof. Suppose not. Let h' be the header targeted by the backjump. As π is cycle free, by Lemma 33, $h \in \mathcal{L}(h')$. By assumed well nesting, we have that $h' \notin \mathcal{L}(h)$. Let π' be the remainder of the path from h' to x. As h' is not within h's body, we must have that π' passes the header h to reach x, but this implies π has a cycle, a contradiction. □

Lemma 35. *If for some a and b such that $succ^+(a, b) \wedge b \notin \mathfrak{D}^+(a)$ and b is not within a loop with a as a header, then:*

1. $\mathfrak{a}$ *is contained within a loop, i.e.* $\exists h.a \in \mathcal{L}(h)$

2. b *is within a loop that also contains* a, *i.e.,* $\exists h'.\{b, a\} \subseteq \mathcal{L}(h')$

3. a *can only reach* b *by following a back jump and entering a loop body containing* b, *i.e.,*

$$\forall \pi . a \xrightarrow{\pi} b \Rightarrow$$
$$\exists i.\text{doms}(\pi(i+1), \pi(i)) \wedge |\pi| > i+1 \wedge \big(\forall j > i+1.\pi(j) \in \mathcal{L}(\pi(i+1))\big)$$

Proof.

1. As b is not in a's loop, then $b \notin \mathfrak{D}^+(a) \wedge b \notin \mathcal{L}(a) \Rightarrow b \notin (succ^+_\downarrow(a) \cup \mathcal{H}(a))$. However, as $b \in succ^+(a)$, all paths from a to b must traverse a back jump. Thus, by Lemma 33 a must be within a loop.

2. Let π be a loop free path from a to b. As $b \notin \mathfrak{D}^+(a)$, there must be at least one backjump along all paths from a to b (otherwise we would have $b \in succ^+_\downarrow(a) \Rightarrow b \in \mathfrak{D}^+(a)$ contradicting our assumption). Let h be the target of the final back jump encountered on π. By Lemma 33, $a \in \mathcal{L}(h) \Rightarrow h \in D(a)$. As there are no further back jumps from h to b, we have $b \in succ^+_\downarrow(h)$. If we have that $b \in D(h)$, we would have $b \in \mathfrak{D}^+(a)$ contradicting our assumption. $b \in succ^+_\downarrow(h) \wedge b \notin \mathfrak{D}(h)$ can only be true if $b \in \mathcal{L}(h)$, i.e., $\mathcal{H}(a) \cap \mathcal{H}(b) \neq \emptyset$.

3. An immediate corollary of the above. Note that the suffix of the path from h to b lies entirely within $\mathcal{L}(h)$. If it did not, we would have an execution that could leave $\mathcal{L}(h)$ and then re-enter without traversing the header h, an impossibility (recall that π is cycle free, and h has been already been visited).

$\square$

Lemma 36. *If for some* a *and* b *such that* $succ^+(a, b) \wedge b \notin \mathfrak{D}^+(a)$, *then either:*

1. *The conditions of Lemma 35 apply, OR*

2. a *is the header of a loop containing* b*, and reaching* b *requires entering the loop of* a.

In either case, if $succ^+(a, b) \wedge b \notin \mathfrak{D}^+(a)$*, then to reach* b *from* a *requires entering a loop containing* b*, i.e.*

$$
\begin{aligned}
&succ^+(a, b) \wedge b \notin \mathfrak{D}^+(a) \Rightarrow \\
&\qquad \forall \pi. a \xrightarrow{\pi} b \Rightarrow \\
&\qquad\qquad \exists i, h. |\pi| > i \wedge \pi(i) = h \wedge b \in \mathcal{L}(h) \wedge \forall j > i. \pi(j) \in \mathcal{L}(h)
\end{aligned}
$$

Proof. If reaching b from a always requires traversing a backjump, then b cannot be within a's loop (by Lemma 34), and Lemma 35 applies.

Otherwise, for a to acyclically reach b and still have $b \notin \mathfrak{D}^+(a)$ requires $b \in \mathcal{L}(a)$. Then it is immediate that reaching b from a requires entering a's loop body. □

We now prove that our scheduling algorithm enforces loop consistency.

Lemma 37. *Suppose we are scheduling an exection of a thread paused at a loop header* h *with* $\mathfrak{I}_h = 0$*. Let* p *be some paused execution point such that there is a path from* p *to* h*. Then there exists some point* q *along* any *path from* p *to* h *such that* $q \in D^+(h)$.

Proof. As h is being scheduled, we must have that $h \notin \mathfrak{D}^+(p)$. But as p reaches h, we have by Lemma 36 that to reach h from p, execution must enter a loop containing h by passing through a loop header h'. However, as $h \in L(h') \Rightarrow h' \in H(h) \Rightarrow h' \in \mathfrak{D}^+(h)$ as required. □

Corollary 37.1. *If execution of a loop* h *begins and there is some paused thread at* p *where* $p \xrightarrow{\pi} h$*, then* $\forall b \in \mathcal{L}(h). \exists i. \pi(i) \in \mathfrak{D}^+(b)$.

Proof. By Lemma 37, there exists some $h' \in \mathfrak{D}^+(h)$ along the path from p to h. As $b \in \mathcal{L}(h)$, we have $h \in \mathfrak{D}^+(b)$, whence $h' \in \mathfrak{D}^+(b)$. □

Corollary 37.1 implies that during execution of loop ℓ no thread outside ℓ may reach ℓ's header.

Lemma 38. *During execution of a loop with header* h*, if a thread within the loop exits the loop, it may not return to the loop header while the loop is still executing.*

Proof. Let $e, e \notin \mathcal{L}(h) \wedge e \neq h$ be the first join point reached after exiting the loop. Then $h \in \mathfrak{D}^+(e)$. To see why, consider the start point of the thread that exits the loop. If it was some $b \in \mathcal{L}(h), b \neq h$ observe that for any member of a loop body, there is a backjump free path from h to b (by Lemma 34). We therefore consider the path from h to e. If the path was backjump free, then $e \in succ^+_\downarrow(h)$, and as $e \notin \mathcal{L}(h)$, we have $e \in \mathfrak{D}^+(h)$. Otherwise, if it was a backjump, then by Lemma 33 e is the header to a loop containing h, whence $e \in \mathfrak{D}^+(h)$. Further, we have $\forall b' \in \mathcal{L}(h).h \in \mathfrak{D}^+(b') \Rightarrow e \in \mathfrak{D}^+(b')$. Thus, the thread at e may not be scheduled until the loop is completed, and by extension may not reach h. □

Together, Corollary 37.1 and Lemma 38 imply that during loop execution no thread with $\mathfrak{I}^h = \iota$ may reach the loop header; i.e., any path of execution that could reach the header of the loop from outside the loop body will eventually be blocked by threads either at the loop header or within the body.

We can now prove the following property:

Theorem 19. *For some loop consistent execution state with threads* $t_1, \ldots, t_n$*, if* t_i *can be scheduled and steps, the resulting state will be loop consistent.*

Proof. Call the scheduled point p. Let the loop vector at this point be $\mathfrak{I}_p$. After stepping, the system will take $n > 0$ paths to a $m \leqslant n$ join points. Let $\mathfrak{I}_t$ be the resulting loop vector produced along one such path π, $\mathfrak{I}_q$ be an unstepped loop vector at point q, and $\mathfrak{I}_u$ be another result loop vector produced along a different path π'. When transforming $\mathfrak{I}_p$ into $\mathfrak{I}_t$ along π, the value for an arbitrary header h could have been transformed according to the following (mutually exclusive) situations:

Entering & Exiting h**'s loop** π originated at h, entered the loop, and then exited it. Then $\mathfrak{I}_t^h = \iota$ and $\mathfrak{I}_t^h \sim \mathfrak{I}_u^h \sim \mathfrak{I}_q^h$ trivially.

Entering h Then $\mathfrak{I}_t^h = \mathfrak{I}_p^h + 1$ and $p = h$. Consider $\mathfrak{I}_q^h$. As the source state is consistent, $\mathfrak{I}_q^h \sim \mathfrak{I}_p^h$, whence $\mathfrak{I}_q^h = \iota$ or $q \in \mathcal{L}(h) \Rightarrow h \in \mathfrak{D}^+(q)$. In the former case, the property holds trivially, in the latter case we could not have stepped. Now consider $\mathfrak{I}_u^h$. If π' entered then exited the loop, or skipped the loop (see below), then $\mathfrak{I}_u^h = \iota$ as required. Otherwise, π' must have entered the loop without exiting, whence $\mathfrak{I}_u^h = \mathfrak{I}^h + 1$ as required. Finally, note that if π or π' terminated at h (i.e., the thread looped back around to the header) the above reasoning holds as the counter is not reset on a backjump.

Skipping h Then π began at the header for h but then skipped the loop, reaching some program point not in the loop body. Then $\mathfrak{I}_t^h = \iota$, giving a trivial result. Note that if π skipped the loop, it cannot also hit the loop header (see below), as this would imply a loop back to h that is not within h's body.

Exiting h Then $\mathfrak{I}_t^h = \iota$, giving a trivial result. As above, π may not exit a loop and then hit the header, as this would imply there exists a path from the body of the loop which exits the loop and returns to the loop header without hitting an intervening join point, an impossibility.

Back jump to h Then $\mathfrak{I}_t^h = \mathfrak{I}_p^h$, $\mathfrak{I}_t^h \neq 0$, $\mathfrak{I}_p^h \neq \iota$. By assumption $\mathfrak{I}_p^h \sim \mathfrak{I}_q^h$ whence $\mathfrak{I}_t^h \sim \mathfrak{I}_q^h$. Consider $\mathfrak{I}_u^h$. As π neither entered or skipped the loop, $p \neq h$, whence π' could not have entered or skipped the loop either. π' may not have hit the header h (see below) as this would imply that π' either began outside the loop or left the loop and returned to h without hitting a join point (both impossibilities). Thus, π' may only have backjumped, hit a join point within h, or exited the loop, all of which give $\mathfrak{I}_u^h = \mathfrak{I}_q^h$ preserving the result trivially.

Hit h's header Then π reached the header of h's loop from outside the body of the loop and $\mathfrak{I}_t^h = 0$. Then $\mathfrak{I}_p^h = \iota$ and $p \neq h$ and $p \notin \mathcal{L}(h)$. Observe that $\mathfrak{I}_q^h = 0 \vee \mathfrak{I}_q^h = \iota$. Otherwise, the loop h is actively executing, and the point p could not have been scheduled (as argued above). Thus $\mathfrak{I}_t^h \sim \mathfrak{I}_q^h$. Finally, π' hits h and we have $\mathfrak{I}_u^h = 0 \sim \mathfrak{I}_t^h$, or π' does not hit h and $\mathfrak{I}_u^h = \iota \sim \mathfrak{I}_t^h$.

No effect Then $\mathfrak{I}_t^h = \mathfrak{I}_p^h = \mathfrak{I}_h^q$. It remains to show that $\mathfrak{I}_h' \sim \mathfrak{I}_h''$. As π did not enter or skip h, we may conclude that $p \neq h$. Then π' either exits h (giving a trivial result), backjumps to h (giving $\mathfrak{I}_u^h = \mathfrak{I}_p^h$ as required), hits h (which is consistent by the reasoning in the above case), or π' does not have any effect on h, whence $\mathfrak{I}_u^h = \mathfrak{I}_p^h$, trivially satsifying the result.

As the inital state with a single thread at method entry with all loop counters set to ι is trivially loop consistent, the above is an inductive argument that any state yielded by our scheduling algorithm will be loop consistent. □

Theorem 20. *From a loop consistent state, if there exists a loop consistent schedule such that a thread* t *paused at* q *may eventually join with a thread* t' *paused at* p *without violating loop consistency, then* $q \in \mathfrak{D}^+(p)$.

Proof. Suppose not. Then q may reach p but $q \notin \mathfrak{D}^+(p)$. As p may reach q, we have that $p \in \mathit{succ}^+(p)$, whence Lemma 36 applies, so there exists some h such that $p \in \mathcal{L}(h)$ and any path from q to p traverses h. By assumption, t must be able to reach p without violating loop consistency. Let our initial loop vector state at q in thread t be $\mathfrak{I}_q$, and the initial state at p be $\mathfrak{I}_p$. Further, $\mathfrak{I}_q^h \sim \mathfrak{I}_p^h \wedge \mathfrak{I}_p^h > 0$.

Consider now the effect on the loop vector of t when it traverses h for the final time (by Lemma 36 t must traverse h at least once). If t reached h from outside the loop, then $\mathfrak{I}_p^h$ will have been set to 0, violating loop consistency (as $\mathfrak{I}_p^h > 0$). Then the final traversal must originate from within the body of the loop, and the final traversal occurs when t has a live counter for h. But then t cannot traverse the header h and enter the loop

body, as this will increment the loop counter for h. Then either the source counter is inconsistent with $\mathfrak{I}_p^h$ or the result counter will be. Thus we conclude that t *cannot* reach p without violating loop consistency, a contradiction. □

Theorem 20 implies that our scheduling algorithm will never schedule a thread for execution if another thread *could* reach it without violating loop consistency. By the same argument, a thread cannot advance past a potential join point until the scheduling algorithm system statically proves that no live threads may reach it.